Operating Systems: A Linux Kernel-Oriented Approach

(Partially written. Expect grammatical mistakes
and minor technical errors.
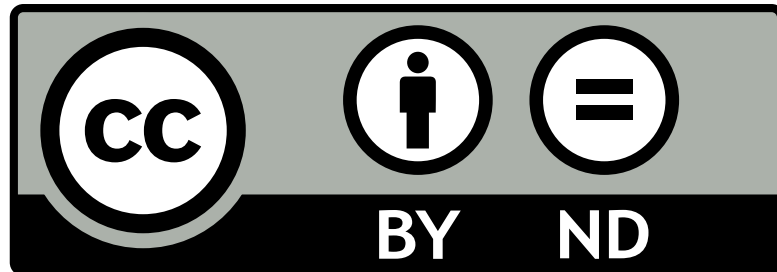Updates are released every week on Fridays.)
Send bug reports/suggestions to
srsarangi@cse.iitd.ac.in
Version 0.93

Smruti R. Sarangi

May 20, 2025

List of Trademarks

- Linux is a registered trademark owned by Linus Torvalds.

- Intel, Intel SGX and Intel TDS are registered trademarks of Intel Corporation.

- AMD is a registered trademark of AMD corporation.

- Microsoft and Windows are registered trademarks of Microsoft Corporation.

- Android, Chrome and Chrome OS are registered trademarks of Google LLC.

- The Unix trademark is owned by the Open Group.

- Red Hat is a registered trademark of Red Hat Inc.

- Suse is a registered trademark of Suse Linux AG.

- Ubuntu and Canonical and registered trademarks of Canonical Ltd.

- webOS is a registered trademark of LG Electronics Inc.

- Tizen is a registered trademark of The Linux Foundation.

- FreeBSD is a registered trademark of The FreeBSD Foundation.

- NetBSD is a registered trademark of The NetBSD Foundation, Inc.

- VMware vSphere is a registered trademark of VMware, Inc.

- Oracle VirtualBox is a registered trademark of Oracle Corporation.

- XenServer is a registered trademark of Citrix Systems, Inc.

- Docker is a trademark of Docker, Inc.

- Podman is a trademark of Podman, Inc.

Contents

Chapter 1

Introduction

Welcome to the exciting world of operating systems. An operating system – commonly abbreviated as an *OS* – is the crucial link between hardware and application programs. We can think of it like a class monitor whose job is to manage the rest of the students. It is a special program, which exercises some control over hardware and other programs. In other words, it has special features and powers that enable it to manage all aspects of the underlying hardware and also ensure that a convenient interface is provided to high-level application software. They should be able to seamlessly operate oblivious of the idiosyncrasies of the underlying hardware.

Let us begin our journey by asking a question, "What is the need for having a specialized program for interacting with hardware and also managing the normal C/Java/Python programs that we write?"

We need to start out with understanding that while designing hardware, our main goals are power efficiency and high performance. Providing a convenient interface to programs is not a goal and neither it should be. It is best to focus on one thing at a time. Moreover, we do not want normal programs to have access to all the features of the underlying hardware because of security concerns and also because any otherwise benevolent, inadvertent change can actually bring the entire system down. Hence, there is a need for a dedicated mechanism to deal with hardware and to also ensure that any operation that potentially has security implications or can possibly bring the entire system down, is executed in a very controlled fashion. This is where the role of the OS becomes important.

Figure 1.1 shows the high-level design of a simple computer system. We can see the CPUs, the memory and the storage/peripheral devices. These are, broadly speaking, the most important components of a modern hardware system. An OS needs to manage them and also needs to make it very easy for a regular program to interact with these entities.

The second figure (Figure 1.2) shows the place of the OS in the overall system. We can see the underlying hardware, high-level programs running on top of it and the OS that sits in the middle. It acts as a mediator, a broker, a security manager and an overall resource manager.
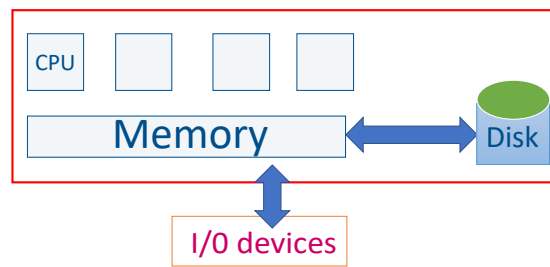
Figure 1.1: Diagram of the overall system
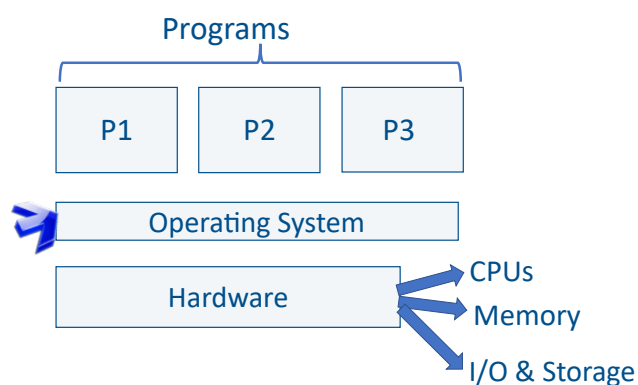
Figure 1.2: Place of the OS in the overall system

Summary 1.0.1

- Programs share hardware such as the CPU, the memory and storage devices. These devices have to be fairly allocated to different programs based on user-specified priorities. The job of the OS is to do a fair resource allocation.

- There are common resources in the system, which multiple programs may try to access concurrently. There is a need to regulate this process such that concurrent accesses are disciplined. It should not be possible for one resource to be used concurrently by multiple running programs when that was not the original intention.

- Different devices have different methods and protocols for managing them. It is essential to speak their language and ensure that high-level commands are translated to device-level commands. This responsibility cannot be put on normal programs. Hence, we need specialized programs within the OS (device drivers) whose job is to exclusively interact with devices.

- Managing the relationships between programs and shared resources such as the memory is fairly complex. For instance, we can have

many running programs that are trying to access the same set of memory locations. This may be a possible security violation or this may be a genuine shared memory-based communication pattern. There is a need to differentiate between them by providing neat and well-defined mechanisms.

- Power, temperature and security concerns have become very important over the last decade. Any operating system that is being designed today needs to run on very small devices such as mobile phones, tablets and even smartwatches. In the foreseeable future, they may run on even smaller devices such as smart glasses or devices that are embedded within the body. Hence, it is important for an OS to be extremely power-aware.

Definition 1.0.1 Definition of an OS

An operating system (OS) works as a CPU manager, memory manager, device manager and storage manager. Its job is to arbitrate accesses to these resources and ensure that programs execute securely, and their performance is maximized subject to power and temperature constraints.

1.1 Types of Operating Systems

We can have different kinds of operating systems based on the target hardware. For instance, we can have operating systems for large high-performance machines. In this case, they would be optimized to execute a lot of scientific workloads and also participate in distributed computing scenarios. On the other hand, operating systems for desktop/laptop machines need to keep the requirements for general-purpose users in mind. It is expected that regular users will use OSes for running programs such as web browsers, word processors, email clients and for watching videos or playing games. Given the highly heterogeneous nature of such use cases, there is a need to support a large variety of programs and also a large variety of devices. Hence, in this case more flexibility is desirable. This increases the susceptibility to viruses and malware. Hence, security is a first-order concern as of today.

The next important usage scenario for an operating system is a mobile device. Nowadays, almost all mobile devices starting from phones to tablets have an operating system installed. For all practical purposes, a mobile device is a full-fledged computer, albeit with reduced hardware resources. Additionally, a mobile phone operating system such as Android® needs to be extremely efficient in terms of both power and performance. The reason is that we don't have a lot of resources available and battery capacity is a major constraint. As a result, the focus should be on optimizing battery life yet providing a good quality of experience.

Operating systems are also making their way into much smaller devices such as smartwatches. Here, we don't expect a lot of applications, but we expect the few applications that run on such watches to operate seamlessly. We expect

that they will work under severe power constraints and deliver a good user experience. Moreover, in this case the code size and the memory footprint of the OS needs to be very small.

1.2   The Linux OS

In this book, we will teach generic OS concepts in the context of the Linux® OS. As compared to all other operating systems, Linux has a very different history. It has not been written by one particular person or one particular organization. In fact, it is a modern marvel in the sense that it has arisen out of a massive worldwide collaboration comprising a very large number of individuals who would otherwise not have been connected with each other. This is truly a remarkable effort and a great example of people coming together to create something that is beneficial to all. Given that the code is open source, and the OS itself is freely available, it has now found widespread acceptance in all kinds of computing platforms ranging from smartwatches to laptops to high-end servers.

It all started in 1990 with Linus Torvalds, a student in Helsinki, Finland, who wanted to work on a freely available version of a variant of the then popular UNIX operating system (Minix). Given the fact that most versions of UNIX® those days were proprietary and were beyond the reach of students, he decided to create an operating system for his Intel-based machine that was a rival of Minix, which was primarily developed and meant to be used in an academic setting. Over the next few years, this activity attracted a lot of developers for whom this was more of a hobby than a profession. All of them contributed either in terms of code or in other ways such as testing the operating system or porting it to new hardware. At the same time, the free software movement was also taking shape. Under the GNU (GNU is not Unix, `https://www.gnu.org/`) umbrella, a lot of software, specifically utilities, were being developed. The Linux developers found a common cause with the GNU developers and developers in the closely-related free software movement (FSF, `https://www.fsf.org/`). As a result, many of the utilities that came to fruition because of these movements got incorporated in the Linux operating system. This was a good fusion of communities, which led to rapid development of the new OS.

Way back in 1992, the first version of Linux was released under the GNU Public License (GPL) [License, 1989]. Believe it or not, the unique nature of the GPL license had a fair amount of impact on the rise and popularity of Linux. It was a free-to-use license similar to many other licenses that were prevalent at that time. Like other free software licenses, it allowed the user to freely download and distribute the code, and make modifications. However, there was an important caveat, which distinguished GPL from other licenses. It was that it is not possible for any redistributing entity to redistribute the code with a more restrictive license. For instance, if let's say someone downloads the Linux code, then it is not possible for her to make proprietary changes and then start selling the OS or even redistribute the modified version without releasing its source code. It is mandatory to release the source code of the modifications under the same GPL license. This ensured that whatever changes and modifications are made to any piece of code that comes with a GPL license still remains the property of the community. Others can use the modifications,

which most likely will be improvements, and then build on them. There were no proprietary walls; this allowed the community to make rapid progress because all incremental improvements had to be shared. However, at that point of time, this was not the case with other pieces of software. Users or developers were not duty-bound to contribute back to the mother repository. This ensured that a lot of the innovations that were made by large research groups and multinational companies were not given back to the community.

Over the years, Linux has grown by leaps and bounds in terms of functionality and popularity. By 2000, it had established itself as a worthy desktop and server operating system. People started taking it seriously and many academic groups started moving away from UNIX to adopt Linux. Given that Linux was reasonably similar to UNIX in terms of the interface and some other high-level design decisions, it was easy to migrate from Unix to Linux. The year 2003 was a pivotal year for the Linux community. This year Linux kernel version 2.6 was released. It had a lot of advanced features and was very different from the previous kernel versions. After this, Linux started being taken very seriously in both academic and industry circles. In a certain sense, it had come of age and had entered the big league. Many companies sprang up that started offering Linux-based offerings, which included the kernel bundled with a set of packages (software programs) and also custom support.

Over the years, Linux distributions such as Red Hat®, Suse® and Ubuntu® (Canonical®) have come to dominate the scene. As of writing this book, circa 2024, they continue to be major Linux vendors. Since 2003, a lot of other changes have also happened. Linux has found many new applications – it has made major inroads into the mobile and handheld market. The Android operating system, which as of 2023 dominates the entire mobile operating space is based on the Linux kernel. Many of the operating systems for smart devices and other wearable gadgets are based on Android. In addition, Google®'s Chrome OS is also a Linux-derived variant. So are other operating systems for Smart TVs such as LG®'s webOS and Samsung®'s Tizen.

Point 1.2.1

It is important to understand the economic model. Especially in the early stages, the GPL licensing model made a lot of difference and was very successful in single-handedly propelling the Linux movement. We need to understand that Linux carved a niche of its own in terms of performance roughly a decade after the project began. The reason it was grown and sustained by a large team of developers in the first formative decade is that they saw a future in it. This also included large for-profit companies. The financial logic behind such extraordinary foresight is quite straightforward.

Why should a hardware company invest in creating a proprietary operating system, when its primary business is processors or software services? It also does not make a lot of sense to buy hundreds of thousands of licenses of a proprietary operating system for its employees and customers. The costs can be prohibitive. It is a much better idea to voluntarily contribute a few employees to the Linux effort such that they create something that is the property of the entire community. This means that

> instead of devoting hundreds of engineers for developing and maintaining a homemade operating system, only tens of engineers are required to ensure that there is a version of Linux that suits the company's needs. If a lot of companies and non-profit groups get together, then the large team is as good as a dedicated OS development team. It can produce something of the same or even a superior quality, at a fraction of the cost. Given that everything is free and users have full control, there is no long-term business risk!

Linux is not the only free open-source operating system. There are many others, which are derived from classical UNIX, notably BSD Unix (Berkeley Standard Distribution) family of operating systems. Some other important variants are FreeBSD®, OpenBSD and NetBSD®. Akin to Linux, their code is also free to use and distribute. Of course, they follow a different licensing mechanism, which is not as restrictive as GPL. However, they are also very good operating systems in their own right. They have their niche markets, and they have a large developer community that actively adds features and ports them to new hardware. A paper by your author and his student S. S. Singh [Singh and Sarangi, 2020] nicely compares three operating systems – Linux, FreeBSD and OpenBSD – in terms of the performance across different workloads.

1.2.1 Versions, Statistics and Conventions

In this book, we will be primarily teaching generic OS concepts. However, it is our firm belief that every operating system concept needs to be explained in the light of a solid practical implementation. This is where code snippets from the latest version of the Linux kernel (as of 2023) will be used. Specifically, we shall use kernel version v6.2 to explain OS concepts. All the OS code that we shall show will be from the main branch. It is available at `https://elixir.bootlin.com/linux/v6.2.12/source/kernel`.

Let us now go through some key code-related statistics of the Linux kernel version v6.2 that has roughly 23 million lines of source code. Every version change typically adds 250,000 lines of source code. The numbering scheme for the kernel version numbers is shown in Figure 1.3.
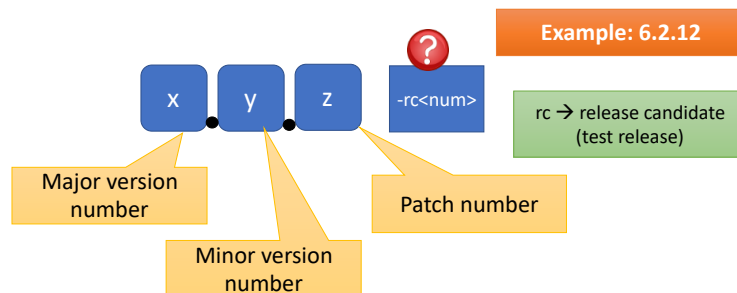
Figure 1.3: Rationale behind assigning Linux kernel versions

Linux Versions

Consider Linux version 6.2.12. Here 6 is the major version number, 2 is the minor version number and 12 is the patch number. Every $\langle major, minor \rangle$ version pair has multiple patch numbers associated with it. A major version represents important architectural changes. The minor version adds important bug fixes and feature additions. A patch mostly focuses on minor issues and security-related bug fixes. Every time there is an important feature-related commit, a patch is created. Prior to 2004, even minor versions were associated with stable versions and odd minor versions were associated with development versions. Ever since Linux kernel version 3.0, this practice has not been adhered to. Every version is stable now. Development versions are now release candidates that predate stable versions.

Every new patch is associated with multiple release candidates. A release candidate does not have major bugs; it incorporates multiple smaller fixes and feature additions that are not fully verified. These release candidates are considered experimental and are not fully ready to be used in a production setting. They are numbered as follows -rc1, -rc2, ... They are mainly aimed at other Linux developers, who can download these release candidates, test their features, suggest improvements and initiate a process of (mostly) online discussion. Once, the discussions have converged, the release candidates are succeeded by a stable version (read patch or major/minor version).

Details of the Linux Code Base
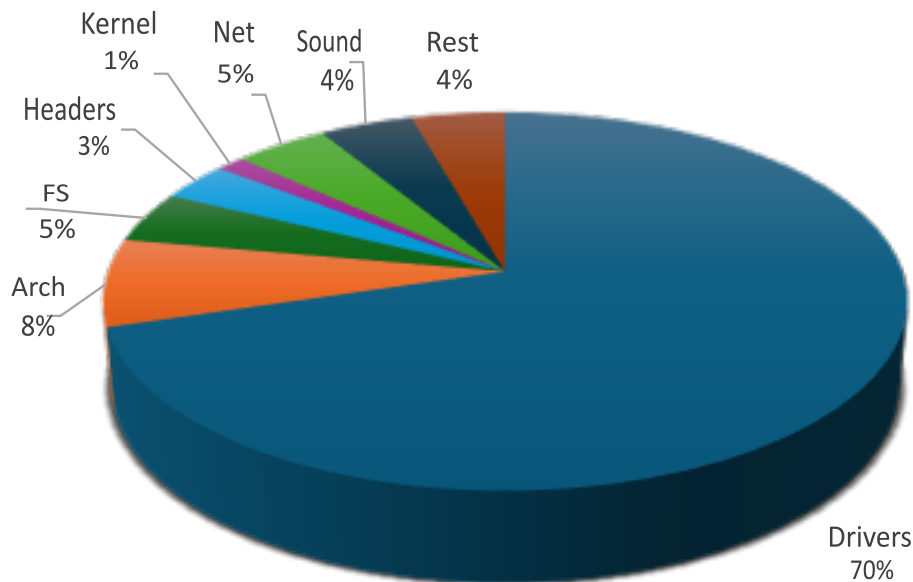
Figure 1.4: Breakup of the Linux code base

Let us now provide an overview of the Linux code base (see Figure 1.4). The

architecture subsystem of the kernel contains all the code that is architecture specific. The Linux kernel has a directory called *arch* that contains various subdirectories. Each subdirectory corresponds to a distinct architecture such as x86, ARM, Sparc, etc. An OS needs to rely on processor-specific code for various critical actions like booting, device drivers and access to privileged hardware operations. All of this code is nicely bundled up in the *arch* directory. The rest of the code of the operating system is independent of the architecture. It is not dependent on the ISA or the machine. It relies on primitives, macros and functions defined in the corresponding *arch* subdirectory. All the operating system code relies on these abstractions such that developers do not have to concern themselves with details of the architecture such as whether it is 16-bit or 32-bit, little endian or big endian, CISC or RISC. This subsystem contains more than 1.7 million lines of code.

The other large subsystems that contain large volumes of code are the code bases for the filesystem and network, respectively. Note that a popular OS such as Linux needs to support many file systems and network protocols. As a result, the code base for these directories is quite large. The other subsystems for the memory and security modules are comparatively much smaller.
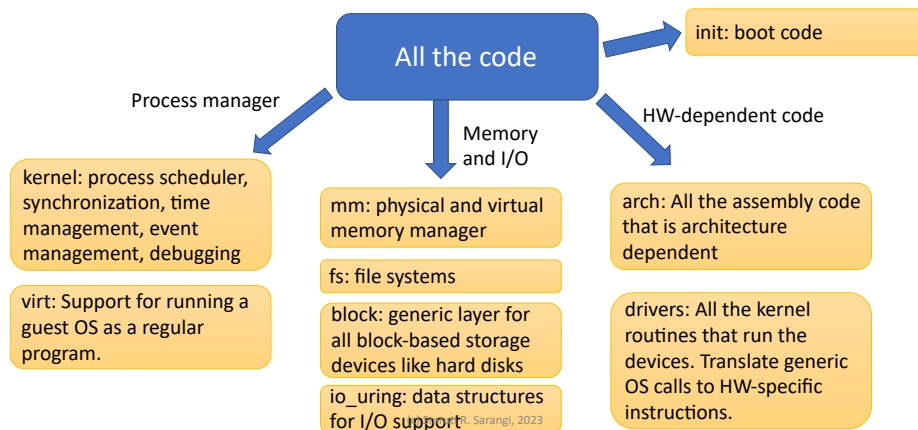
Figure 1.5: Important directories in the Linux kernel's code base

Figure 1.5 shows the list of prominent directories in the Linux kernel. The *kernel* directory contains all the core features of the Linux kernel. Some of the most important subsystems are the scheduler, time manager, synchronization manager and debugging subsystem. It is by far the most important subsystem – it is the core of the kernel. We will focus a lot on this subsystem.

We have already seen the *arch* directory. A related directory is the *init* directory that contains all the booting code. Both these directories are hardware dependent.

The *mm*, *fs*, *block* and *io_uring* directories contain important code for the memory subsystem, file system and I/O modules, respectively. The code for virtualizing an operating system is resident in the *virt* directory. Virtualizing the OS means that we can run an OS as a regular program on top of the Linux OS. This subsystem is tightly coupled with the memory, file and I/O subsystems.

Finally, note that the largest directory is *drivers* that contains drivers (spe-

cialized programs for talking to devices) for a large number of I/O devices. This directory is so large because an operating system such as Linux needs to support a large amount of hardware. For every hardware device, we should not expect the user to browse the web, locate its driver and install it. Hence, there is a need to include its code in the code base of the kernel itself. At the same time, we do not want to include the code of every single device driver on the planet in the code base of the kernel. Its code will become prohibitively large. Rarely used and obsolescent devices can be left out. Hence, the developers of the kernel need to judiciously choose the set of drivers that need to be included in the kernel's code base, which is released and distributed. These devices should be reasonably popular, and the drivers should be deemed to be safe (devoid of security issues).

1.3   Organization of the Book
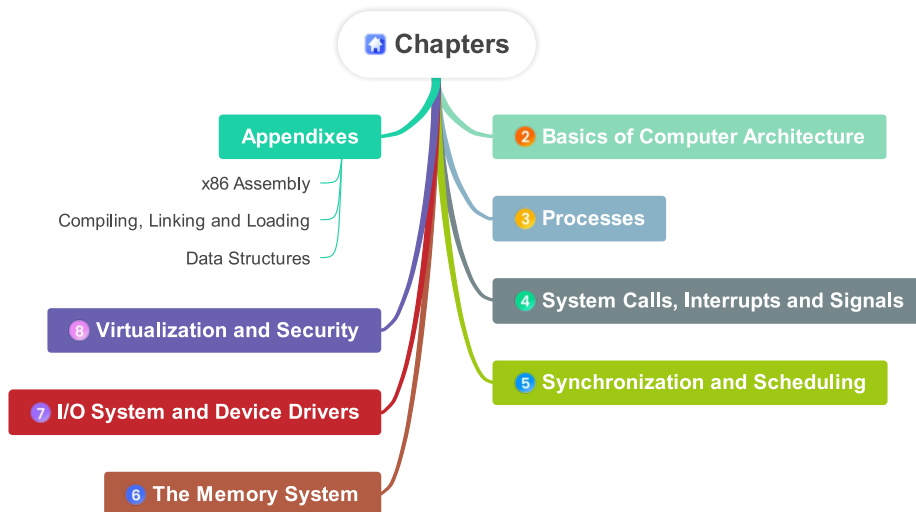
Figure 1.6: List of chapters

Figure 1.6 shows the list of chapters and appendixes in the book. All the chapters use concepts that may require the user to refer to the appendixes. There are three appendixes in the book. Appendix A introduces the x86 assembly language (the 64-bit variant). We shall refer to snippets of assembly code throughout the text. To understand them thoroughly, it is necessary to be familiar with x86 assembly. Most of the critical routines in operating systems are still written in assembly language for speed and efficiency. Appendix B describes the compiling, linking and loading process. This appendix should be read thoroughly because it is important to understand how large C-based software projects are structured. Readers should know the specific roles of C files, header files, .o files, static and dynamically linked libraries. These concepts are described in detail in this chapter. Finally, Appendix C introduces the most commonly used data structures in the Linux kernel. A lot of the data structures that we typically study in a basic undergraduate data structures course have

scalability problems. Hence, it is necessary to use specialized data structures in the Linux kernel. We shall specifically introduce generic linked list containers, red-black trees, B and B+ trees, maple trees, radix trees and augmented trees. These are used in the Linux kernel to solve sophisticated problems. Let us now provide a brief overview of each chapter.

We shall start with an overview of the computer architecture concepts needed to undertake a study of operating systems. Recall that the primary function of an OS is to abstract the architecture and provide a convenient interface to high-level applications. Hence, it is very important to understand the nature of the underlying hardware. It is what we are building an OS for. The three important subsystems that we shall cover are the cores including their registers and interrupt-processing hardware, virtual memory and the I/O system. Virtual memory, especially, plays a very important role in the design of operating systems. It is mostly managed by the OS and is needed to enforce isolation across processes (running instances of programs). Finally, note that for writing device driver code and for realizing the I/O subsystem, an understanding of I/O devices and the interfaces needed to access them is required. We shall cover all these topics and finally move to quintessential OS concepts.

The third chapter is on processes. Our focus will be on the data structures used by the kernel to store the state of processes. We shall observe that data structure design is a key challenge. There are important trade-offs involved. In some cases, trees are more scalable and in some cases hash tables are more scalable. Sometimes we need to use a combination of data structures to enhance efficiency. Once the `task_struct` structure that represents a process's state is fully understood, we shall move on to understanding the mechanisms of process creation and destruction. Linux has a special method of creating new processes – existing processes are cloned and if needed their runtime image is replaced. Finally, we shall look at the process of context switching, i.e., switching between processes. If one process is running it, we need to pause it and resume another process that has been waiting to execute. This mechanism is the backbone of any multitasking system and in practice is an intricate choreography of different small operations that need to be executed in a precise sequence. Any process can potentially be paused and resumed thousands of times unbeknownst to it – its runtime state needs to be restored exactly. We shall see that doing this with 100% reliability is an engineering marvel.

Next, we shall look at methods to communicate information to and from processes using a host of mechanisms. Accessing any OS function is not as simple as making a function call. The address spaces of the application and the OS kernel are different. Hence, invoking an OS function and passing arguments to it is a non-trivial task. This involves a context switch also. We shall start with looking at system calls, which are methods for user programs to seek OS intervention. Making a system call is an elaborate process, which has a very well-defined convention for passing arguments and accessing return values. Next, we shall look at interrupting events such as hardware interrupts of software-generated exceptions. Both interrupt the running process and start executing an OS routine to handle the interrupt. We shall look at accurately saving the state of the running process and executing a very high-priority interrupt handler in this section. We shall also answer complex questions of the form, "What happens when an interrupt arrives when another interrupt is being processed?" (case of nested interrupts) In Linux, interrupt processing happens in two stages:

the urgent work is completed immediately and the rest of the work is completed later. There are different types of kernel tasks that can do such *deferred work*. They run with different priorities and have different features. Specifically, we shall introduce softirqs, threaded IRQs and work queues. Finally, we shall introduce signals, which are the reverse of system calls. The OS uses signals to send messages to running processes. For example, if we press a mouse button, then a message goes to the running process regarding the mouse click and its coordinates. This happens via signals. Here also there is a need to change the context of the user application because it now needs to start processing the signal.

Chapter 5 is a long chapter on synchronization and scheduling. In any modern OS, we have hundreds of running *tasks* that often try to access shared resources concurrently. Many such shared resources can only be accessed by one thread at a time. Hence, there is a need for synchronizing the accesses. This is known as locking in the context of operating systems. Locking is a large and complex field that has a fairly strong overlap with advanced multiprocessor computer architecture. We specifically need to understand it in the context of memory models and data races. Memory models determine the valid outcomes of concurrent programs on a given architecture. We shall observe that it is often necessary to restrict the space of outcomes using special instructions to correctly implement locks. If locks are correctly implemented and used, then uncoordinated accesses known as data races will not happen. Data races are the source of a lot of synchronization-related bugs. Once the basic primitive has been designed, we shall move on to discussing different types of locks and advanced synchronization mechanisms such as semaphores, condition variables, reader-writer locks and barriers. The kernel needs many concurrent data structures such as producer-consumer queues, mutexes, spinlocks and semaphores to do its job. We shall look at their design and implementation in detail.

Next, we shall move on to explaining a very interesting synchronization primitive that is extremely lightweight and derives its correctness by stopping task preemption at specific times. It is known as the read-copy-update (RCU) mechanism, which is widely used in the kernel code. It is arguably one of the most important innovations made by the designers of the kernel, which has had far-reaching implications. It has obviated the need for a garbage collector. We shall then move on to discussing scheduling algorithms. After a cursory introduction to trivial algorithms like shortest-job first and list scheduling, we shall move on to algorithms that are actually used in the kernel such as completely fair scheduling (CFS). This discussion will segue into a deeper discussion on real-time scheduling algorithms where concrete guarantees can be made about schedulability and tasks getting a specific pre-specified amount of CPU time. In the context of real-time systems, another important family of algorithms deal with locking and acquiring resources exclusively. It is possible that a low-priority process may hold a resource for a long time while a high-priority process is waiting for it. This is known as *priority inversion*, which needs to be avoided. We shall study a plethora of mechanisms to avoid this and other problems in the domain of real-time scheduling and synchronization.

Chapter 6 discusses the design of the memory system in the kernel. We shall start with extending the concepts that we studied in Chapter 2 (architecture fundamentals). The role of the page table, TLB, address spaces, pages and folios will be made clear. For a course on operating systems, understanding these

concepts in exquisite detail is necessary. We shall start with classical schemes and move on to the way page management is done in the Linux kernel. The most important concepts are reverse mapping and the MGLRU page aging and replacement algorithm. Reverse mapping is defined as the process of mapping physical frames to virtual pages. This is important because the same frame may be mapped to the address spaces of multiple processes. We thus need to keep track of this information and update it when processes are forked. The MGLRU page aging and replacement algorithm is a game-changing innovation. It is a novel approach to managing the state associated with a large number of pages in a very scalable fashion. We shall do a thorough code-level analysis. The chapter will conclude with a look at the different kernel-level memory allocators. Note that we do not have a page table for a reasonably large part of the kernel's address space. Therefore, it is necessary to manage physical memory directly and thus special memory allocators are necessary.

Chapter 7 introduces I/O systems, storage devices, device drivers and file systems. We need to understand that the I/O stack is very intricately connected with the design of the devices themselves. Hence, a good OS designer needs to understand the details of the devices such that the software stack can use their features efficiently and cover up for their deficiencies. In fact, it is necessary to understand the hardware in a reasonable amount of detail otherwise the device drivers will fall short of their desired performance targets. Hence, we shall first look at commonly used I/O and storage devices used as of 2024 in great detail. Once, we understand their relative pros and cons, we shall proceed to understand the structure of devices and I/O requests in Linux. This is a common layer that individual device drivers build on. Linux defines two kinds of devices: block devices and character devices. The former type of devices are typically storage devices such as flash memories and hard disks that read and write large blocks in one go. Their device drivers are typically built in to the kernel and have a quite elaborate structure. Given that they need to transfer a large amount of data, it is important to minimize the latency and maximize the throughput. On the other hand, character devices like mice and keyboards are not that latency sensitive. However, for them the ease of use and installation is quite important. There are a plethora of such devices. It should be easy to write drivers for them and integrate them easily into a running system.

The final chapter deals with security and virtualization (Chapter 8). Today, security is a first-order design criterion. We shall start this chapter with a discussion of different kinds of access control methods in modern operating systems. Security policies can be specified at the level of resources such as file, or they can be specified at the level of users and groups. We will discuss specific technologies that have proven to be quite useful such as SELinux, AppArmor, PAM and extended file attributes. They help specify fine-grained security policies. We shall then discuss security modules in the latest version of the Linux kernel. We shall specifically focus on how they monitor accesses and restrict some aspects of user behavior that are deemed to be too risky. Finally, we will touch upon the kernel's cryptographic API and auditing infrastructure.

The second part of the chapter deals with *virtualization*. Virtualization allows a guest operating system to run on top of Linux as a regular application. For example, we can run three instances of Linux and two instances of Windows on a Linux machine as if they were normal processes. Each operating system will operate in isolation and presume that it is actually running on a real machine.

Basically, the CPU and the devices are being virtualized here. As of today, virtualization and its lightweight version namely containers are the most popular technologies in the cloud computing ecosystem. Some popular virtualization software are VMWare vSphere®, Oracle VirtualBox® and XenServer®. They are also known as *hypervisors*. Linux has a built-in hypervisor known as Linux KVM (kernel virtual machine). We will study more about them in this chapter. We will also look at lightweight virtualization techniques using containers that virtualize processes, users, the network, file systems, configurations and devices. Docker® and Podman® are important technologies in this space. In the last part of this chapter we shall look at specific mechanisms for virtualizing the I/O system and file systems, and finally conclude.

Exercises

Ex. 1 — What are the roles and functions of a modern operating system?

Ex. 2 — Is a system call like a regular function call? Why or why not?

Ex. 3 — Why is the *drivers* directory the largest directory in the kernel's code base?

Ex. 4 — What are the advantages of having a single *arch* directory that stores all the architecture-specific code? Does it make writing the rest of the kernel easier?

Ex. 5 — Write a report about all the open-source operating systems in use today. Trace their evolution.

Ex. 6 — Discuss the implications of the GPL license on the development of Linux.

Ex. 7 — Do you think the C language is the right choice for the Linux kernel?

Chapter 2

Basics of Computer Architecture

An operating system is the connecting link between application programs and hardware. Hence, it is essential that any serious student of operating systems gains a fair understanding of programming languages and the way programs are written, and the way hardware is designed (computer architecture). The aim of this chapter is to outline the fundamentals of computer architecture that are needed to understand the working of an operating system. This chapter does not aim to teach the student computer architecture in its entirety. The student is requested to consult traditional textbooks on computer architecture [Sarangi, 2021, Sarangi, 2023] for getting a deeper understanding of all the concepts. The aim of this chapter is to provide an overview such that the student has sufficient opportunity for recapitulation and can get a clearer understanding of some key hardware features that modern OSes rely on.

Organization of the Chapter

Figure 2.1 shows the organization of this chapter. The main aim of this chapter is to cover all the computer architecture concepts needed to understand modern operating systems. The objective is not to explain well-known computer architecture concepts such as cores, caches and the memory system. The focus is only on specific hardware features that are relevant for understanding a book on operating systems.

We shall start with looking at the privileged mode of execution, which operating systems use. This is normally not taught in regular computer architecture courses because regular user programs cannot access privileged registers and privileged instructions. We need to look at such instructions because they are very useful for writing software such as the Linux kernel. Privileged registers can be used for controlling the underlying hardware such as turning off the display or the hard disk. Next we shall discuss methods to invoke the OS and application-OS communication. No OS program normally runs. The kernel (core part of the OS) begins to run only when there is an event of interest: the system boots, a hardware device raises an interrupt, there is a software bug such as an illegal access or the running program raises a dummy software interrupt to get the attention of the OS kernel. If interrupts are not naturally being generated, then there is a need to create dummy interrupts using a timer chip – a
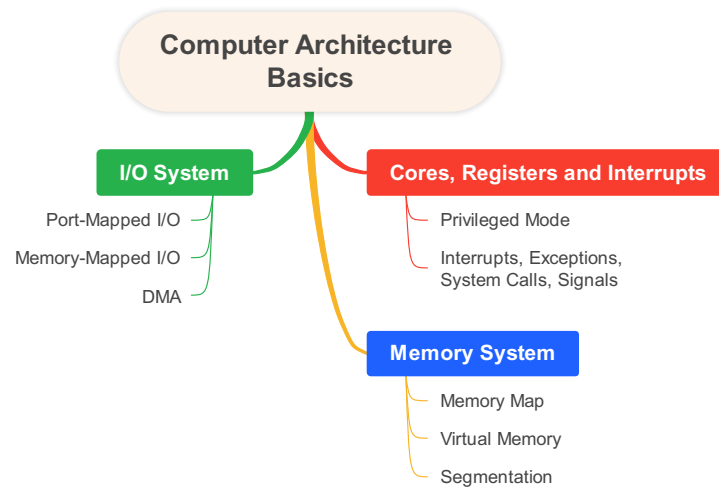
Figure 2.1: Organization of this chapter

programmable, periodic interrupt generator.

Next, we shall discuss the details of the memory system, especially the virtual memory system, which is relevant from an OS perspective. We need to first understand how a process views its memory space. It assumes that it on an n-bit machine, it can access most of the 2^n addressable bytes. Furthermore, it assumes a fixed structure for the entire memory space. This is known as the memory map. This is a very elegant assumption and makes it easy for programmers to write code, compilers to generate code and processors to run instructions. Given that processors run multiple processes at the same time, there is a need to map this abstract view of memory (virtual memory) to a memory addressing mechanism on a real machine. This is done using the memory management unit of the processor and relevant OS code. This process involves the use of hardware structures like the TLB and software structures like the page table. Intel® and AMD® processors that use the x86 ISA further complicate the situation by using segment registers where the entire virtual address space can be split it into multiple segments. A hardware-based segment register maintains the starting address for each segment.

Finally, we shall look at the hardware support for I/O instructions and devices. The simplest approach is to create a set of I/O registers and assign them to I/O devices. Reading or writing to these registers is tantamount to reading or writing to the I/O device. This classical method is known as port-mapped I/O. It is primarily meant for low-bandwidth devices. For faster devices that consume more data at a time, it is a better idea to share a large memory region with them. The underlying hardware ensures that any write to this memory region by a device driver is equivalent to transferring the entire memory region to the device. Reads work in a similar fashion, albeit the direction of the flow of data is in the other way. This is known as memory-mapped I/O. Both these

approaches require the active involvement of the CPU. The third approach relies on outsourcing this work to DMA (Direct Memory Access) engines that often reside outside the chip. They do the entire job of transferring data to or from the I/O device. Once the transfer is done, they raise an interrupt.

After reading this entire chapter, the reader will have sufficient knowledge in specific aspects of computer architecture that are relevant from an OS perspective. The reader is strong encouraged to also go through Appendixes A and B. They cover the x86 assembly language and an introduction to the process of compiling, linking and loading, respectively. We shall continuously be referring to concepts discussed in these appendixes. Hence, it makes a lot of sense to go through them after completing this chapter.

2.1 Cores, Registers and Interrupts

2.1.1 Multicore Systems
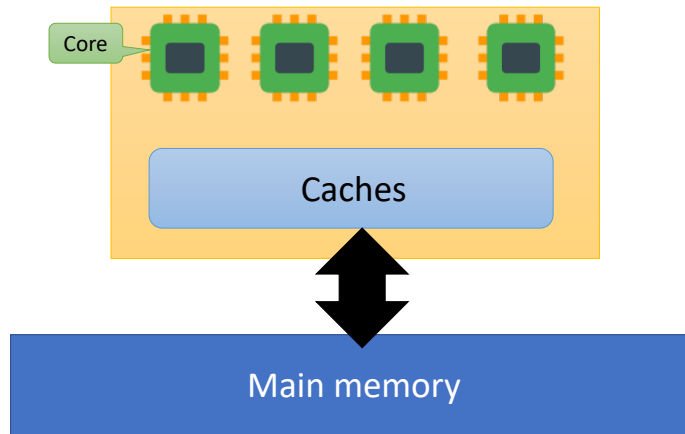
Figure 2.2: A multicore processor

Figure 2.2 shows the structure of a typical multicore processor. As of 2024, a multicore processor has anywhere between 4-64 cores, where each core is a fully functional pipeline. Furthermore, it has a hierarchy of caches. Each core typically has an instruction cache (i-cache) and a data cache (d-cache or L1 cache). These are small yet very fast memories ranging from 8 KB to 64 KB. Then, we have an L2 cache and possibly an L3 cache as well, which are much larger. Depending upon the type of the processor, the L3 cache's size can go up to several megabytes. Some recent processors (as of 2024) have started to include an additional L4 cache as well. However, that is typically on a separate die housed in the same multichip module, or on a separate layer in a 3D chip (refer to the design of the Intel Meteorlake CPU [Zimmer et al., 2021]).

The last level of the cache (known as the LLC) is connected to the main memory (via memory controllers), which is quite large – 8 GB to 1 TB as of 2024. Needless to say it is the slowest of all the elements in the memory hierarchy. It is typically made up of DRAM memory cells that are slow yet have

a very high storage density. The most important point that we need to keep in mind here is that it is only the main memory – DRAM memory located outside the chip – that is visible to software, notably the OS. The rest of the smaller memory elements within the chip such as the L1, L2 and L3 caches are normally not visible to the OS. Some ISAs have specialized instructions that can flush certain levels of the cache hierarchy either fully or partially. Sometimes even user applications can use these instructions. However, this is the only notable exception. Otherwise, we can safely assume that almost all software including privileged software like the operating system are unaware of the caches. Let us live with the assumption that the highest level of memory that an OS can see or access is the main memory.

Let us define the term *memory space* as the set of all addressable memory locations. A software program including the OS perceive this memory space to be one large array of bytes. Any location in this space can be accessed at will and also can be modified at will. Later on when we discuss virtual memory, we will refine this abstraction.

2.1.2   Inside a Core

Let us now take a sneak peek inside a core. A core is a fully-featured pipeline, which can either be a regular in-order pipeline or an out-of-order pipeline. Furthermore, each core has some amount of cache memory: level 1 instruction and data caches. The core also has a set of named storage locations that are accessible by instructions directly; they are known as *registers*. A typical processor has 8-32 registers. The advantage of having registers is that they can be accessed very quickly by instructions, often in a fraction of a cycle. All the registers are stored in a register file, which is made up of SRAMs; it is significantly faster than caches that typically take multiple cycles to access.

Most of the operations in a core happen on the registers. Registers are often both the operands in an instruction. Even when a location in memory needs to be accessed, the memory address is computed based on values stored in registers. For instance, in the 32-bit x86 ISA, the expression `mov %eax, 4(%esp)` stores the value in the `eax` register into the memory location whose address is as follows. The base address A is stored in the %esp register and the offset is 4. The memory address is equal to $(A+4)$. Given the speed and ease of access, registers are ubiquitous. They are additionally used to access privileged locations and I/O addresses, as we shall see later.

Definition 2.1.1 CISC and RISC ISAs

A Reduced Instruction Set Computer (RISC) has a simple and regular ISA. It needs to use a lot more registers as compared to its competitor, i.e., CISC ISAs. A CISC (Complex Instruction Set Computer) has a large and diverse collection of instructions that are often more complex than a RISC ISA's simple and regular set of instructions. The advantage of a RISC ISA is the simplicity of its decoder and code generation algorithms. Whereas, CISC ISAs excel on machines where the aim is to reduce the number of bytes used to encode instructions.

Next, let us differentiate between CISC and RISC processors. RISC stands

for "Reduced Instruction Set Computer". A lot of the modern ISAs such as ARM and RISC-V are RISC instruction sets, which are regular and simple. RISC ISAs and processors tend to use registers much more than their CISC (complex instruction set) counterparts. CISC instructions can have long immediates (constants) and may also use more than one memory operand. The instruction set used by Intel and AMD processors, x86, is a CISC ISA. Regardless of the type of the ISA, registers are central to the operation of any program (be it RISC or CISC). The compiler needs to manage them efficiently.

2.1.3 Registers

General Purpose Registers

Let us look at the space of registers in some more detail. All the registers that regular programs use are known as *general purpose registers*. They are visible to all software including the compiler. Note that almost all the programs that are compiled today use registers and the author is not aware of any compilation model or any architectural model that does not rely on registers.

Privileged Registers

A core also has a set of registers known as *privileged* registers, which only the OS or software with similar privileges can access. In Chapter 8, we shall look at hypervisors or virtual machine managers (VMMs) that run with OS privileges. All such software are known as system software or privileged mode software. They are given special treatment by the CPU – they can access privileged registers.

For instance, an ALU has a *flags* register that stores its state, especially the state of instructions that have executed in the past such as comparison instructions. Often these *flags* registers are not fully visible to regular application-level software. However, they are visible to the OS and anything else that runs with OS privileges such as VMMs. It is necessary to have full access to these registers to enable multitasking: run multiple programs on a core one after the other.

We also have control registers that can enable or disable specific hardware features such as the fan, LED lights on the chassis and can even turn off the system itself. We do not want all the instructions that change the values stored in these registers to be visible to regular programs because then a user application can create havoc. Hence, we entrust only a specific set of programs (OS and VMM) with access rights to these registers.

Then, there are debug registers that are meant to debug hardware and system software. Given the fact that they are privy to additional information and can be used to extract information out of running programs, we do not allow regular programs to access these registers. Otherwise, there will be serious security violations. However, from a system designer's point of view or from the OS's point of view these registers are very important. This is because they give us an insight into how the system is operating before and after an error is detected – this information can potentially allow us to find the root cause of bugs.

Finally, we have I/O registers that are used to communicate with externally placed I/O devices such as the monitor, printer and network card. Here again, we need privileged access. Otherwise, we can have serious security violations,

and different applications may try to monopolize an I/O resource. They may not allow other applications to access them. Hence, the OS needs to act as a broker. Its job is to manage, restrict and regulate accesses.

Given the fact that we have discussed so much about privileged registers, let us see how the notion of privileges is implemented. Note that we need to ensure that only the OS and related system software such as the VMM can have access to privileged resources such as the privileged registers.

Current Privilege Level Bit

Along with registers, modern CPUs also store the current mode of execution. For instance, they need to store the state/mode of the current CPU, which basically says whether it is executing operating system code or not. This is because if it is executing OS code, then we need to allow the executing code to access privileged registers and also issue privileged instructions. Otherwise, if the CPU is executing normal application-level code, then access to these privileged registers should not be allowed. Hence, historically, processors always have had a bit to indicate the status of the program that they are executing, or alternatively the mode that they are in. This is known as the *Current Privilege Level* or *CPL* bit. In general, a value equal to zero indicated a privileged mode (the OS is executing) and a value equal to one indicated that an application program is executing.
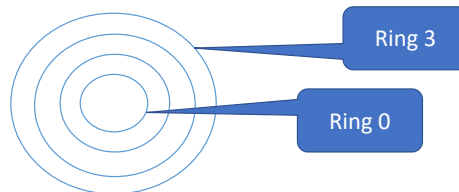
Figure 2.3: Rings in x86 processors

Modern-day processors typically have more modes of execution. Intel processors, for instance, have four modes of execution, which are also known as *rings* – Ring 0 (OS) to Ring 3 (application) (refer to Figure 2.3). The primary role of rings 1 and 2 is to run guest operating systems (OSes as regular applications) and other software that do not require as much of privileged access as the software running in ring zero. Nevertheless, they enjoy more privileges than regular application code. As mentioned earlier, they are typically used while running guest OSes on virtual machines, where a *virtual machine* is defined as a software environment that emulates the functionality of a multicore CPU.

Privileged and Non-Privileged Instructions

Most instructions are non-privileged. This means that they are regular load-/store, arithmetic, logical and branch instructions. These instructions can be executed by all types of code including the application and the OS. These instructions can also seamlessly execute when the OS is executing as a guest OS on a virtual machine.

Recall that we also discussed privileged instructions, when we discussed privileged registers. These are specialized instructions that allow the program to change the internal state of the processor like changing its frequency or accessing certain features that a regular program should never have access to. These include control registers, debug registers and I/O registers.

We will ask an important question here and answer it when we shall discuss virtual machines in Chapter 8. What happens when application code or code running at a lower privilege level (higher ring) accesses instructions that should be executed by code running at a higher privilege level (lower ring)? In general, we would expect that there will be an exception. Then the appropriate exception handler can take over and take appropriate action. If this is the case, we shall see in Chapter 8 that writing a virtual machine is reasonably easy. However, there are a lot of instructions in the instruction sets of modern processors that do not show this behavior. Their behavior is far more confusing and pernicious. They either remain silent (like a nop) or yield different results when executed in different modes without generating exceptions. We shall see that handling such instructions is quite difficult and that is why the design of virtual machines is actually quite challenging. The main reason for this is that when instruction sets were initially created, virtual machines were not around and thus designers could not think that far. As a result, they thought that having such polymorphic instructions (instructions that change their behavior based on the ring level) was a good idea. When virtual machines started gaining prevalence, this turned out to be a huge problem, as we shall see later.

2.1.4 Interrupts, Exceptions, System Calls and Signals

The discussion in this chapter up till now should have convinced the reader that an application program in itself is quite incompetent. For instance, it does not have access to large parts of the hardware and also does not have a lot of control on its own execution or the execution of other processes. Hence, there is a necessity to actively engage with the underlying operating system. There are different ways by which the operating system and application communicate. Let us quickly go through them.

Interrupts An interrupt is a specialized message sent to the processor via an I/O device or its associated controller, which corresponds to an external hardware event such as a key press or the arrival of a network packet. In this case, it is important to draw the attention of the CPU such that it can process the interrupt. This would entail stopping the execution of the currently executing program and jumping to a memory location that contains the code of the *interrupt handler* (specialized routine in the OS to handle the interrupt).

Exception An exception corresponds to an error in the execution of the program. This could be an event such as dividing by zero, issuing an illegal instruction or accessing an address that is not mapped to main memory. In this case, an exception is generated, which is handled by its corresponding exception handler (part of the OS code).

System Call If an application needs some service from the OS such as creating a file or sending a network packet, then it cannot use the conventional

mechanism, which is to make a function call. OS functions cannot be directly invoked by the application. Hence, there is a need to generate a dummy interrupt to garner the attention of the OS. In this case, a specialized system call handler takes over and satisfies the request made by the application.

Signal A system call is a message that is sent from the application to the OS. A signal is the reverse. It is a message that is sent from the OS to the application. An example of this would be a key press. In this case, a hardware interrupt is generated, which is processed by the OS. The OS reads the key that was pressed, and then figures out the process that is running in the foreground. The ASCII value of this key needs to be communicated to this process. The signal mechanism is the method that is used. In this case, a function registered by the process with the OS to handle a "key press" event is invoked. The running application process then gets to know that a certain key was pressed and depending upon its logic, appropriate action is taken. A signal is basically a *callback function* that an application registers with the OS. When an event of interest happens (pertaining to that signal), the OS calls the callback function in the application context. This callback function is known as the *signal handler*.

As we can see, communicating with the OS does require some novel and unconventional mechanisms. Traditional methods of communication that include writing to shared memory or invoking functions are not used because the OS runs in a separate address space and also switching to the OS is an onerous activity. It also involves a change in the privilege level and a fair amount of bookkeeping is required at both the hardware and software levels, as we shall see in subsequent chapters.

Example of a System Call

Let us provide a brief example of a system call. It is in fact quite easy to issue, even though application developers are well advised to not directly issue system calls mainly because they may not be sure of the full semantics of the call. Furthermore, operating systems do tend to change the signature of these calls over time. As a result, code that is written for one version of the operating system may not work for a future version. Therefore, it is definitely advisable to use library calls like the standard C library (`glibc`), which actually wrap the system calls. Library calls almost never change their signature because they are designed to be very flexible. Flexibility is not a feature of system calls because parameter passing is complicated. Consequently, library calls remain portable across versions of the same operating system and also across different variants of an operating system such as the different flavors of Linux.

In the file arch/x86/entry/syscalls/syscall_64.tbl, a maximum of 548 system calls can be defined. For kernel v6.2, roughly 362 calls are defined for 64-bit architectures and 36 calls are defined for 32-bit architectures. The standard way to make a system call is as follows.

```
mov $<sys call number>, %rax
syscall
```

As we can see, all that we need to do is that we need to load the number of the system call in the `rax` register. The `syscall` instruction subsequently does the rest. It generates a dummy interrupt, stores some data corresponding to the state of the executing program (for more details, refer to [Sarangi, 2021]) and loads the appropriate system call handler. An older approach is to directly generate an interrupt itself using the instruction `int 0x80`. Here, the code 0x80 stands for a system call. However, as of today, this method is not used for x86 processors.

Saving the Context

The state of the running program is known as its *context*. Whenever, we have an interrupt, exception or a system call, there is a need to store the context, jump to the respective handler, finish some additional work in the kernel (if there is any), restore the context and start the original program at exactly the same point. The caveat is that all of these actions need to happen without the explicit knowledge of the program that was interrupted. Its execution should be identical to a situation where it was not interrupted by an external event. Of course, if the execution has led to an exception or system call, then the corresponding event/request will be handled. In any case, we need to return to exactly the same point at which the context was switched.
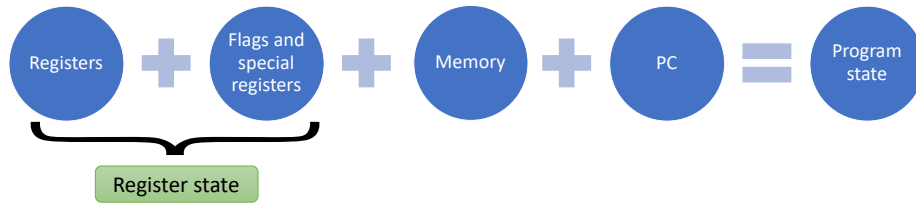
Figure 2.4: The "context save" process

Figure 2.4 shows an overview of the process to store the context of a running program. The state of the running program comprises the contents of the general purpose registers, contents of the flags and special purpose registers, the memory and the PC (program counter). Towards the end of this chapter, we shall see that the virtual memory mechanism stores the memory space of a process very effectively and stops other processes from unintentionally or maliciously modifying it. Hence, we need not bother about storing and restoring the memory contents of a process. It is not affected by the context switch and restore process.

Insofar as the rest of the three elements are concerned, we can think of all of them as the `volatile` state of the program that is erased when there is a context switch. As a result, a hardware mechanism is needed to read all of them and store them in memory locations that are known a priori. We shall see that there are many ways of doing this and there are specialized and privileged instructions that are used.

For more details about what exactly the hardware needs to do, readers can refer to the computer architecture text by your author [Sarangi, 2021]. In the example pipeline in the reference, the reader will appreciate the need for having specialized hardware instructions for automatically storing the PC, the flags and

special registers, and possibly the stack pointer in either privileged registers or a dedicated memory region. Regardless of the mechanism, we have a known location where the volatile state of the program is stored, and it can later on be retrieved by the interrupt handler. For clarity and readability, we will use the term *interrupt handler* to refer to traditional interrupt handlers, as well as exception handlers and system call handlers, whenever the context makes this clear.

Subsequently, the first task of the interrupt handler is to retrieve the program state or context of the executing program – either from specialized registers or a dedicated memory area. Note that these temporary locations may not store the entire state of the program, for instance they may not store the values of all the general purpose registers. The interrupt handler will thus have to do more work and retrieve the full program state. Regardless of the specific mechanism, the role of the interrupt handler is to collect the full state of the executing program and ultimately store it somewhere in memory, from where it can easily be retrieved later.

Restoring the context of a program is quite straightforward. We need to follow the reverse sequence of steps.

The life cycle of a process can thus be visualized as shown in Figure 2.5. The application program executes, it is interrupted for a certain duration after the OS takes over, then the application program is resumed at the point at which it was interrupted. Here, the word "interrupted" needs to be understood in a very general sense. It could be a hardware interrupt, a software interrupt like a system call or a program-generated exception.
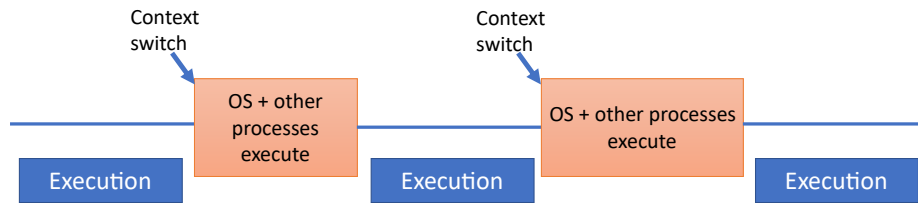
Figure 2.5: The life cycle of a process (active and interrupted phases)

We can comprehend this situation as follows. The OS treats an application as an *atomic entity* that can be moved from core to core, suspended at any point of time and resumed later, possibly on the same core or on a different core. It is a fully self-contained entity that does not carry any baggage from its execution on a previous core (from a correctness point of view). The context save and restore process is thus very effective – it fully saves the running state of the process such that it can be restored at any point of time later (same or different core).

Timer Interrupts

There is an important question to think about here. Consider a system where there are no interrupts and executing processes do not generate system calls and exceptions. Assume that there are n cores, and each core runs such a process that does not lead to system calls or exceptions. This means that the

OS will never get executed because its routines will never get invoked. Note that the operating system never executes in the background (as one would want to naively believe) – it is a *separate program* that needs to be invoked by a very special set of mechanisms namely system calls, exceptions and interrupts. Let us refer to these as *events of interest*. The OS cannot come into the picture (execute on a core) any other way.

Now, we are looking at a very peculiar situation where all the cores are occupied with programs that do none of the above. There are no events of interest. The key question that we need to answer is whether the system becomes unresponsive if these programs decide to run for a long time. Is rebooting the system the only option?

Question 2.1.1

Assume we have a situation, where we have a single-core machine and the program that is running on the core is purely computational in nature. It does not make any system calls, and it also does not lead to any exceptions. Furthermore, assume that there is no hardware or I/O activity, and therefore no interrupts are generated. In such a situation, the process that is running on the core can potentially run forever unless it terminates on its own. Does that mean that the entire system will remain unresponsive till this process terminates? We will have a similar problem on a multicore machine where there are k cores and k regular processes on them, where no events of interest are generated.

This is a very fundamental question in this field. Can we always rely on system calls, exceptions and interrupts (events of interest) to bring in the operating system? It is indeed possible that we have a running program that does not generate any events of interest. In such a situation, when the OS is not running, an answer that the OS will somehow swap out the current process and load another process in its place is not correct. A core can run only one process at a time, and if it is running a regular application process, it is not running the OS. If the OS is not running on any core, it cannot possibly act.

Point 2.1.1

The operating system is in many ways like a regular program that needs a core to run. If no OS process is running at a point of time on any core, then the OS is clearly not functional. It cannot act or take any action. Unless it is invoked in some manner such as an interrupt or a system call, there is no way to run OS code.

We need to thus create a mechanism to ensure that the OS periodically runs regardless of the frequency of events of interest. This mechanism is known as a timer interrupt. As shown in Figure 2.6, there is a separate timer chip on the motherboard that periodically sends timer interrupts to the CPU. There is a need for such a chip because we need to have a *guaranteed source of* interrupts. Whenever, a timer interrupt is generated, it is routed to one of the cores, there is a context switch and the OS starts executing. This is how the OS periodically comes in even when there is no other event of interest. All platforms that support an operating system need to have a timer chip. It is arguably the
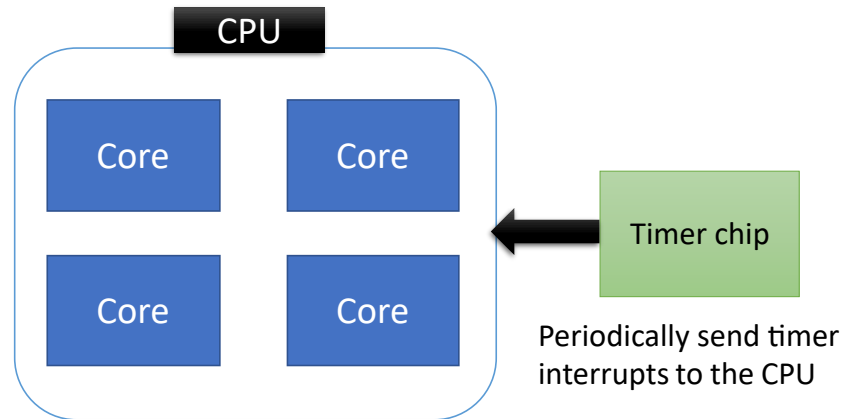
Figure 2.6: The timer chip generates periodic interrupts

most integral part of a machine that supports an operating system. The key insight is that it is needed for ensuring that the system is responsive, and it periodically executes the OS code. The operating system kernel has full control over the processes that run on cores, the memory, storage devices and I/O systems. Hence, it needs to run periodically such that it can effectively manage the system and provide a good quality of experience to users.

Listing 2.1: Jiffies
source : include/linux/jiffies.h

```
extern unsigned long volatile jiffies;
```

We divide time into *jiffies*. A timer interrupt is generated at the end of every *jiffy*. The number of jiffies (jiffy count) is incremented by one when a timer interrupt is received. The duration of a jiffy has been reducing over the course of time. It used to be 10 ms in the Linux kernel around a decade ago and as of 2023, it is 1 ms. It can be controlled by the compile-time parameter HZ. If HZ =1000, it means that the duration of a jiffy is 1 ms. We do not want a jiffy to be too long, otherwise the system will take a fair amount of time to respond. Simultaneously, we also do not want it to be too short, otherwise a lot of time will be spent in servicing timer interrupts.

Inter-processor interrupts

As we have discussed, the OS gets invoked on one core and its subsequent job is to take control of the system and basically manage everything such as running processes, waiting processes, cores, devices and memory. Often there is a need to ascertain if a process has been running for a long time or not and whether it needs to be swapped out or not. If there is a need to swap it out, then the OS finds the most eligible process (using its scheduler) and runs it.

If the new process needs to run on the core on which the OS is executing, then it is simple. All that needs to be done is that the OS needs to load the context of the process that it wants to run. If a process on a different core needs to be swapped out to make room for the selected process, the mechanism

becomes more complex. It is necessary to send an interrupt to the remote core such that an OS process starts running on it. This mechanism is known as an inter-processor interrupt (or IPI). Almost all processors today, particularly all multicore processors, have a facility to send an IPI to any core with support from the hardware's interrupt controller. Relevant kernel routines frequently use such APIs to run the OS on any core. The OS process may choose to do more of management and bookkeeping activities or quickly find the next process to run on the core.

> **Point 2.1.2**
>
> It is easy for the OS kernel to run a user process on the "current" core on which it is executing. However, running a process on a different (remote) core is a more elaborate task. There is a need to send an interrupt to the remote core using the IPI mechanism. This causes a context switch to the OS on the remote core. The OS process that starts on the remote core can then swap in a user process.

2.2   Memory System

How does a process view the memory space? Should it be aware of other processes and the memory regions that they use? Unless we provide an elegant answer to such questions, we will entertain many complex corner cases and managing memory will be very difficult. We are in search of simple abstractions.

One large array of bytes
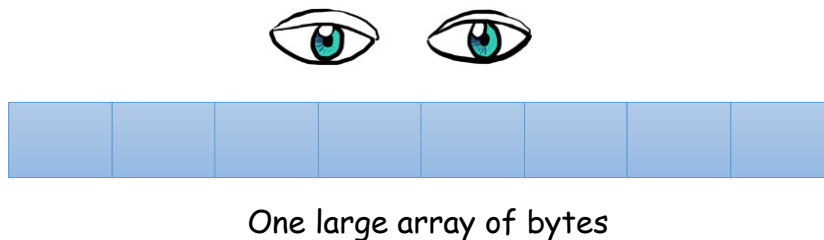
Figure 2.7: The way that a programmer or compiler view the memory space

Such a simple abstraction is shown in Figure 2.7. A process simply views the entire memory space as a sequence of bytes. For instance, in a 32-bit architecture, a process assumes that it can access any of the 2^{32} bytes at will. Similarly, in a 64-bit architecture, a process assumes that it can access any of the 2^{64} bytes at will. The same assumption is made by the programmer and the compiler. In fact, a large part of the pipeline and the CPU also make the same assumption. It is quite difficult to make any other assumption. This assumption is elegant and simplifies the job of everybody other than the engineers designing the memory system.

Trivia 2.2.1

In most modern ISAs, load and store instructions read their base addresses from registers. They add a constant offset to it. The size of registers thus determines the range of addresses that can be accessed. If registers are 32 bits wide, then the size of the address space is naturally constrained to 2^{32} bytes.

Compatibility Problem

As a rule, in an n-bit architecture, where the register size is n bits, we assume that the instructions can access any of the addressable 2^n bytes unless there are specific constraints. The same assumption needs to be made by the programmer and the compiler because they only see the registers. Other details of the memory system are not directly visible to them.

Note that a program is compiled only once on the developers' machines and then distributed to the world. If a million copies are running, then we can be rest assured that they are running on a very large number of heterogeneous devices. These devices can be very different from each other. Of course, they will have to share the same ISA, but they can have radically different main memory sizes and even cache sizes. Unless we assume that all the 2^n addresses are accessible to a program or process, no other elegant assumption can be made. This may sound impractical for 64-bit machines, but this is the most elegant assumption that can be made.

This has a potential to cause problems. For example, if we assumed that a process can access 4 GB at will, it will not run on a system with 1 GB of memory, unless we find a mechanism to do so. We thus have a *compatibility problem* here, where we want our process to assume that addresses are n bits wide (n is typically 32 or 64), yet run on machines with all memory sizes (typically much lower than the theoretical maximum).

Definition 2.2.1 Compatibility Problem

Processes assume that they can access any byte in a hypothetically large memory region of size 2^{32} or 2^{64} bytes at will (for 32-bit and 64-bit systems, respectively). Even if processes are actually accessing very little data, there is a need to create a mechanism to run them on physical machines with far lower memory (let's say a few GBs). The memory addressing scheme is not compatible with the physical memory system of real machines. This is the *compatibility problem*.

Our simplistic assumption allows us to easily write a program, compile it, and also distribute the compiled binaries to run on machines that have memories of all sizes. Subsequently, when a processor runs it, it can also live with the same assumption and assume that the entire address space, which is very large in the case of a 64-bit machine, is accessible to the running program (process). All of this is subject to successfully solving the compatibility problem. There are unfortunately several serious problems that get introduced because of this assumption. The most important problem is that we can have multiple processes

that are either running one after the other (using multitasking mechanisms) or are running in parallel on different cores. These processes can access the same address because nothing prohibits them from doing so.

Overlap Problem

In this case, unbeknownst to multiple processes, they can corrupt each other's state by writing to the same address. One program can be malicious, and then it can easily get access to the other's secrets. For example, if one process stores a credit card number, another process can read it straight out of memory. This is clearly not allowed and presents a massive security risk. Hence, we have two opposing requirements over here. First, we want an addressing mechanism that is as simple and straightforward as possible such that programs and compilers remain simple and assume that the entire memory space is theirs. This is a very convenient abstraction. However, on a real system, we also want different processes to access a different set of addresses such that there is no unintended overlap between the set of memory addresses that they access. This is known as the *overlap problem.*

> **Definition 2.2.2** Overlap Problem
>
> Unless adequate steps are taken, it is possible for two processes to access overlapping regions of memory, and also it is possible to get unauthorized access to other processes' data by simply reading values that they write to memory. This is known as the *overlap problem.*

Size Problem

We are sadly not done with our set of problems; it turns out that we have another serious problem on our hands. It may happen that we want to run a program whose memory footprint is much more than the physical memory that is present on the system. For instance, the memory footprint could be two GBs whereas the total physical memory is only one GB. It may be convenient to say that we can simply deny the user the permission to execute the program on such a machine. However, the implications of this are severe. It basically means that any program that is compiled for a machine with more physical memory cannot run on a machine with less physical memory. This means that it will cease to be backward compatible – not compatible with older hardware that has less memory. In terms of a business risk, this is significant.

Hence, all attempts should be made to ensure that such a situation does not arise. It turns out that this problem is very closely related with the overlap and compatibility problems that we have seen earlier. It is possible to slightly repurpose the solution that we shall design for solving the overlap and compatibility problems.

> **Summary 2.2.1**
>
> We have identified three problems namely the compatibility, overlap and size problems. All of these problems arise when we translate the hypothetical or the virtual view of the user to the addressing mechanisms in

a real system.

2.2.1 Memory Map of a Process

Let us continue assuming that a process can access all memory locations at will. We need to understand how it makes the life of the programmer, compiler writer and OS developer easy. Figure 2.8 shows the memory map of a process in the Linux operating system. The memory map is a layout of the memory space that a process assumes. It shows where different types of data and code are stored.
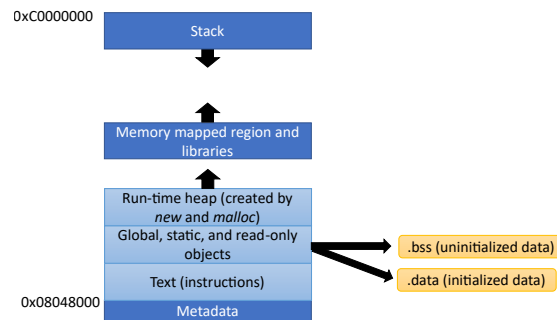
Figure 2.8: The memory map of a process in 32-bit Linux

The memory map is partitioned into distinct sections. It starts from address zero. Then after a fixed offset, the *text* section starts, which contains all the program's instructions. The processor starts executing the first instruction at the beginning of the text section and then starts fetching subsequent instructions as per the logic of the program. Once the text section ends, the *data* section begins. It stores initialized data that comprises global and static variables that are typically defined outside the scope of functions. After this, we have the *bss* (block starting symbol) section that stores the same kind of variables, however they are uninitialized. It is possible that one process has a very small data section and another process has a very large data section – it all depends upon how the program is written.

Then we have the *heap* and the *stack*. The heap is a memory region that stores dynamically allocated variables and data structures, which are typically allocated using the `malloc` call in C and the `new` call in C++ and Java. Traditionally, the heap section has grown upwards (towards increasing addresses). As and when we allocate new data, the heap size increases. It is also possible for the heap size to decrease as we free or dynamically delete allocated data structures. Then there is a massive hole, which basically means that there is a very large memory region that doesn't store anything. Particularly, in 64-bit machines, this region is indeed extremely large.

Next, at a very high memory location (0xC0000000 in 32-bit Linux), the *stack* starts. The stack typically grows downwards (grows towards decreasing addresses). Given the fact that there is a huge gap between the end of the heap and the top of stack, both of them can grow to be very large. If we consider the value 0xC0000000, it is actually 3 GB. This basically means that in a 32-bit

system, an application is given at the most 3 GB of memory. One can argue that if the size of the stack, heap and other sections combined exceeds 3 GB, we shall run out of space. This indeed can happen and that is why the world has transitioned to 64-bit systems, where such problems will not happen.

The last unanswered question is what happens to the one GB region that is remaining (recall 2^{32} bytes = 4 GB)? This is a region that is typically assigned to the operating system kernel for storing all of its runtime state. As we shall see in later chapters, there is a need to split the address space between user applications and the kernel.

Now, the interesting thing is that all processes share the same structure of the memory map. This means that the chances of them destructively interfering with each other is even higher because most variables will have similar addresses: they will be stored in roughly the same region of the memory map. Even if two processes are absolutely innocuous (harmless), they may still end up corrupting each other's state, which is definitely not allowed. As a result, ensuring a degree of separation is essential. Another point that needs to be mentioned with regard to the kernel memory space is that it is an invariant across process memory maps. It always resides in the top one GB of the memory map of every process. This region is out of bounds for regular user processes.

The advantage of having a fixed memory map structure is that it is very easy to generate code. Binaries can also have a fixed format that is in line with the memory map and operating systems know how to layout code and data in memory. Regardless of the elegance, simplicity and standardization, we need to solve the overlap problem. Having a standard memory map structure makes this problem worse because now regardless of the process, the variables are stored in roughly the same set of addresses. Therefore, the chances of destructive interference become very high.

2.2.2 Virtual Memory

Our objective is to basically respect each process's memory map, run multiple processes in parallel if there are multiple cores, and also run several processes one after the other on the same core via the context switch mechanism. To do all of this, we somehow need to ensure that they are not able to corrupt or even access each other's memory regions. Clearly, the simplest solution is to somehow restrict the memory regions that a process can access.

Base and Limit Registers

Let us look at a simple implementation of this idea. Assume that we have two registers associated with each process: *base* and *limit*. The base register stores the first address that is assigned to a process and the limit register stores the last address. Between base and limit, the process can access every memory address. In this case, we are constraining the addresses that a process can access and via this we are ensuring that no overlap is possible. We observe that the value of the base register need not be known to the programmer or the compiler. All that needs to be specified is the difference between the limit and base (maximum number of bytes a process can access).

The first step is to find a *free* memory region when a process is loaded. Its size needs to be greater than or equal to the maximum size specified by the

process. The starting address of this region is set as the contents of the base register. The address sent to the memory system is computed by adding the address computed by the CPU with the contents of the base register. All the addresses computed by the CPU are as per the memory map of the process; however, the addresses sent to the memory system are different. In this system, if the process accesses an address that is beyond the limit register, then a fault is generated. Refer to Figure 2.9 for a graphical illustration of the base-limit scheme.
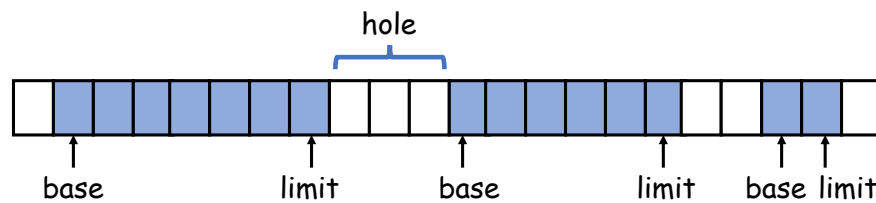
Figure 2.9: The base-limit scheme

We observe that there are many processes, and they have their memory regions clearly demarcated. Therefore, there is no chance of an overlap. This idea does seem encouraging, but this is not going to work in practice for a combination of several reasons. The biggest problem is that neither the programmer nor the compiler know for sure how much memory a program requires at run time. This is because for large programs, the user inputs are not known, and thus the total memory footprint is not predictable. Even if it is predictable, we will have to budget for a very large footprint (conservative maximum). In most cases, this conservative estimate is going to be much larger than the memory footprints we may see in practice. We may thus end up wasting a lot of memory. Hence, in the memory region that is allocated to a process between the base and limit registers, there is a possibility of a lot of memory getting wasted. This is known as *internal fragmentation*.

Let us again take a deeper look at Figure 2.9. We see that there are holes or unallocated memory regions between allocated memory regions. Whenever we want to allocate memory for a new process, we need to find a hole that is larger than or equal to what we need and then split it into an allocated region and a smaller hole. Very soon we will have many such holes in the memory space, which cannot be used for allocating memory to any other process. It may be the case that we have enough memory available, but it is just that it is partitioned among so many processes that we do not have a contiguous region that is large enough. This situation where a lot of memory is wasted in such holes is known as *external fragmentation*.

> **Definition 2.2.3** Fragmentation
>
> Fragmentation means wastage of memory space. It can be of two types: internal and external. While allocating memory, processes are often assigned fixed chunks of memory that cannot be used by other processes. Sometimes some memory space is wasted within a chunk. This is known as *internal fragmentation*.
> There may be regions of memory that are not a part of such allocated chunks. It may not be possible to allocate this memory to processes. This is known as *external fragmentation*.

There are many ways of solving this problem. Some may argue that periodically we can compact the memory space by reading data and transferring them to a new region by updating the base and limit registers for each process. In this case, we can essentially merge holes and create enough space by creating one large hole. The problem is that a lot of reads and writes are involved in this process and during that time the process needs to remain mostly stalled.

Another problem is that the prediction of the maximum memory usage may be wrong. A process may try to access memory that is beyond the limit register. As we have argued, in this case a fault is generated. However, this can be avoided if we allocate another memory region and link the second memory region to the first (using a linked list like structure). The algorithm now is that we first access the memory region that is allocated to the process and if the offset is beyond the limit register, then we access the second read memory region. The second remain memory region will also have base and limit registers. We can extend this idea and create a linked list of such memory regions. We can also save time by having a lookup table. It will not be necessary to traverse linked lists. Given an address, we can quickly figure out the memory region in which it lies. Many of the early approaches focused on such kind of techniques, and they grew to become very complex, but soon the community realized that this is not a scalable solution, and it is definitely not elegant.

Need for Address Translation

However, an important insight came out of this exercise. It was that the address that is generated by the CPU, which is also the same address that the programmer, process and compiler see, is not the address that is ultimately sent to the memory system. Even in this simple case, where we use a base and limit register, the address generated by the program is actually added to the contents of the base register to generate the real memory address. The real or *physical address* is sent to the memory system. The gateway to the memory system is the instruction cache for instructions and the L1 data cache for data. They only see the physical address. On the other hand, the address generated by the CPU is known as the *virtual address*. There is a need to translate or convert the virtual address to a physical address such that we can access memory and solve the overlap problem, as well as the compatibility problem.

Definition 2.2.4 Virtual and Physical Addresses

The virtual address is the address seen by the programmer, process, compiler and the CPU. In a k-bit architecture, it is normally k bits. However, this address is not presented to the memory system. The virtual address is converted or *translated* to a physical address, which is then sent to the memory system. If every physical address is mapped to only one virtual address, then there will never be any overlaps across processes. This approach naturally solves the overlap and compatibility problems.

A few ideas emerge from this discussion. Given a virtual address, there should be a table that we can look up, and find the physical address that it maps to. Clearly, one virtual address will always be mapped to one physical address. This is a common sense requirement. However, if we can also ensure that every physical address maps to only one virtual address across processes (barring special cases), or in other words there is a strict one-to-one mapping, then we observe that no overlaps between processes are possible. Regardless of how hard a process tries, it will not be able to access or overwrite the data that belongs to any other process. In this case we are using the term *data* in the general sense – it encompasses both code and data. Recall that in the memory system, code is actually stored as data.

Way Point 2.2.1

To effectively solve the compatibility and overlap problems, we have two preliminary ideas now. The first is that we assume that the CPU issues virtual addresses, which are generated by the running process. These virtual addresses are then *translated* to physical addresses, which are sent to the memory system. Clearly, we need a one-to-one mapping to prevent overlaps. Furthermore, we desire a mechanism that uses a fast lookup table to map a virtual address to its corresponding physical address.

> **Definition 2.2.5** Virtual Memory
>
> *Virtual memory* is defined as an abstract view of the memory system where a process assumes that it is the exclusive owner of the entire memory system, and it can access any address at will from 0 to $2^n - 1$ in an n-bit memory system. Practical implementations of the virtual memory abstraction solve the compatibility and overlap problems. It is additionally possible to solve the size problem by also including storage devices to expand the available memory space (as we shall see later).
> The method for doing this is to map every virtual address to a physical address. The mapping automatically solves the compatibility problem, and if we ensure that a physical address is never mapped to two different virtual addresses, then the overlap problem is also easily solved. We can always extend the physical address space to comprise not only locations in the main memory but also locations on storage media such as a part of the hard disk. As a result, the physical address space can indeed exceed the size of the main memory.

The crux of the entire definition of virtual memory (see Definition 2.2.5) is that we have a mapping table that maps each virtual address (that is used by the program) to a physical address. If the mapping satisfies some conditions, then we can solve all the three problems. So the main technical challenge in front of us is to properly and efficiently create the mapping table to implement an address translation system.

2.2.3 Address Translation System

Pages and Frames

Let us start with a basic question. Should we map addresses at the byte level or at a higher granularity? To answer this question, we can use the same logic that we use for caches. We typically consider a contiguous block of 64 or 128 bytes in caches and treat it as an atomic unit. This is called a *cache block*. The memory system comprising the caches and the main memory only deals with blocks. The advantage of this is that the memory system remains simple, and we do not have to deal with a lot of entries in the caches. The opposite would have been true if we had addressed the caches at the byte level. Furthermore, owing to temporal and spatial locality, the idea of creating blocks has some inherent advantages. The first is that the same block of data will most likely be used over and over again. The second is that by creating blocks, we are also implicitly prefetching. If we need to access only four bytes, then we actually fetch 64 bytes because that is the block size. This ensures that when we access data that is nearby, it is already available within the same block.

Something similar needs to be done here as well. We clearly cannot maintain mapping information at the byte level – we will have to maintain a lot of information – this is not a scalable solution. We thus need to create blocks of data for the purpose of mapping. In this space, it has been observed that a block of 4 KB typically suits the needs of most systems very well. This block of 4 KB is known as a *page* in the virtual memory space and as a *frame* or a *physical page* in the physical memory space.

> **Definition 2.2.6** Pages and Frames
>
> A page is typically a block of 4 KB of contiguous memory in the virtual memory space. A page can be mapped to a 4 KB block in the physical address space, which is referred to as a physical page or a frame.

Our mapping problem is much simpler now – we need to map 4 KB pages to 4 KB frames. It is not always the case that we have 4 KB pages and frames. In many cases, especially in large servers, it is common to have huge pages that as of 2023 can be from 2 MB to 1 GB. In all cases, the size of a page is equal to the size of the frame that it is mapped to.

An example mapping is shown in Figure 2.10. Here we observe that dividing memory into 4 KB chunks has proven to be really beneficial. We can create the mapping in such a way that there are no overlaps. Addresses that are contiguous in virtual memory need not be contiguous in physical memory. However, if we have a fast lookup table then all of this does not matter. We can access any virtual address at will and the mapping system will automatically convert it to a physical address, which can be accessed without the fear of overlaps or any other address compatibility issues. We have still not brought in solutions for the size problem yet. But we shall see later that it is very easy to extend this scheme to incorporate additional regions within storage devices such as the hard disk into the physical address space.
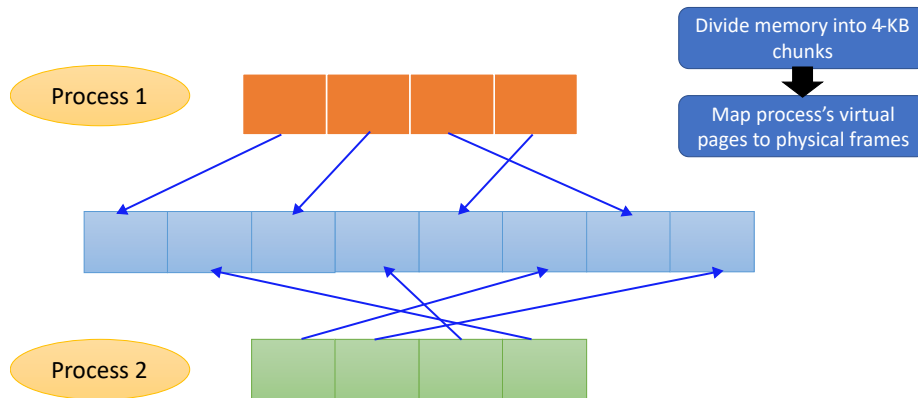
Figure 2.10: Conceptual overview of the virtual memory based page mapping system

The Page Table

Let us refer to the mapping table as the *page table*. Let us first explain in the context of a 32-bit memory system. Each page is 4096 bytes or 2^{12} bytes. We thus require 12 bits to address a byte within a page, and we need 20 (32-12) bits to specify a page address. We will thus have 2^{20} or roughly a million pages in the virtual address space of a process. For each page, we need to store a 20-bit physical frame address. The total storage overhead is thus (20 bits = 2.5 bytes) multiplied with one million, which turns out to be 2.5 MB. This is the

storage overhead per process, because every process needs its own page table. Now assume that we have 100 processes in the system, we therefore need 250 MB to just store page tables !!!

This is a prohibitive overhead. If we consider a 64-bit memory system, then the page table storage overhead is even larger and clearly this idea will not work. It represents a tremendous wastage of physical memory space. Let us thus propose optimizations. To start with, note that most of the virtual address space is actually not used. In fact, it is quite sparse particularly between the stack and the heap. This region can actually be quite large (refer to Section 2.2.1). Recall that the beginning of the virtual address space is populated with the text, data, bss and heap sections. Then there is a massive hole between the heap and the stack. The stack starts at the upper boundary of the virtual address space. In some cases, memory regions corresponding to memory mapped files and dynamic libraries can occupy a part of this region. We shall still have large gaps and have a significant amount of sparsity. This insight can be used to design a multilevel page table, which can leverage this pattern and prove to be a far more space-efficient solution.
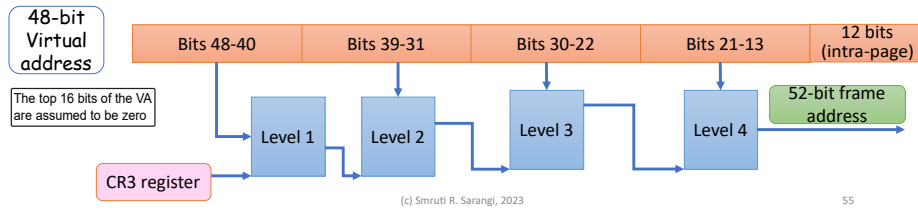
Figure 2.11: The design of the multilevel page table

The design of a multilevel page table is shown in Figure 2.11. It shows an example address translation system for a 64-bit machine. We typically realize that we don't need that large a virtual address space. 2^{64} bytes is more than a billion gigabytes, and no practical system (as of 2024) has that much of memory space. Hence, most practical systems as of 2024, use a 48-bit virtual address. This is sufficient. The top 16 (MSB) bits are assumed to be zero. We can always break this assumption and have more levels in a multilevel page table. This is seldom required. Let us thus proceed assuming a 48-bit virtual address. We however assume a full 64-bit physical address in our examples. Note that the physical address can be as wide as possible because we are just storing a few additional bits per entry – we are not adding new levels in the page table (as we shall observe). Given that 12 bits are needed to address a byte in a 4 KB page, we are left with 52 bits. Hence, a physical frame number is specified using 52 bits. Figure 2.12 shows the memory map of a process assuming that the lower 48 bits of a memory address are used to specify the virtual memory address.

In our 48-bit virtual address, we use the bottom 12 bits to specify the address of the byte within the 4 KB page. Recall that 2^{12} bytes = 4 KB. We are left with 36 bits. We partition this group of bits into four blocks of 9 bits each. If we count from 1, then these are bit positions 40-48, 31-39, 22-30 and 13-21. Let us consider the topmost level, i.e., the top 9 bits (bits 40-48). We expect the least amount of randomness in these bits. The reason is obvious. In any system with temporal and spatial locality, we expect most addresses to be close by. They may

2^{48} - 1

Stack

Memory mapped region and libraries

Run-time heap (created by *new* and *malloc*)

Global, static, and read-only objects

.bss (uninitialized data)

.data (initialized data)

Text (instructions)

0x400000

Metadata
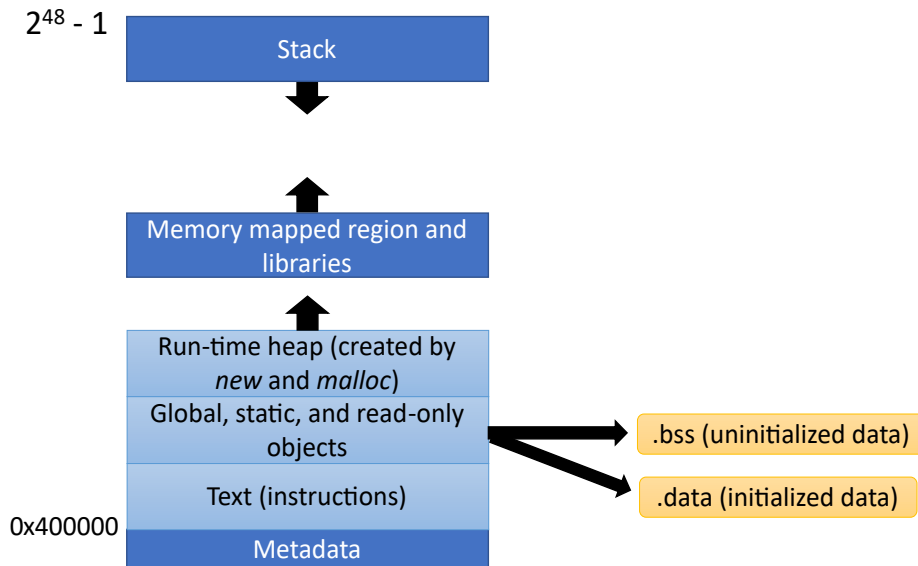
Figure 2.12: The memory map in 64-bit Linux

vary in their lower bits, however, in all likelihood their more significant bits will be the same. To cross-check, count from 0 to 999 (in base 10). The unit's digit changes the most frequently. It changes with every number. The ten's digit on the other hand changes more infrequently. It changes after every 10 numbers, and the hundred's digit changes even more infrequently. It changes once for every 100 numbers. By the same logic, when we consider binary addresses we expect the more significant bits to change far less often than the less significant bits. Given this insight let us proceed to design an optimized version of the page table.

Let us consider the first set of 9 bits (bits 40-48). They can be used to access 2^9 (=512) entries in a table. Let us create a Level 1 page table that is indexed using these 9 bits. An entry in this table is either null (empty) or points to a Level 2 page table. Given our earlier explanation about the structure of the memory map, we expect most of the entries in the Level 1 page table to be null. This will happen because the memory map is sparse and each set of top-level 9 bits points to a large contiguous region. Most of these regions will be unallocated. This means that we shall have to allocate space for very few Level 2 page tables. There is no need to allocate a Level 2 page table if the set of addresses that it corresponds to are all unallocated. For example, assume that there are no allocated virtual addresses whose top 9 bits (bits 40-48) are equal to the binary sequence 011010100. Then, the corresponding row in the L1 page table will store a null value and no corresponding Level 2 page table will be allocated. This is the key insight that allows us to save space.

Note that we need to store the address of the Level 1 page table somewhere. This is typically stored in a machine specific register on Intel hardware called the CR3 register. Whenever a process is loaded, the address of its Level 1 page table is loaded into the CR3 register. Whenever, there is a need to find a mapping in the page table, the first step is to read the CR3 register and find

the base address of the Level 1 page table. It is a part of the process's context.

We follow a similar logic at the next level. The only difference is that in this case there may be multiple Level 2 page tables. Unlike the earlier case, we don't need to store their starting addresses in dedicated registers. The Level 1 entries point to their respective base addresses. We use the next 9 bits to index entries in the Level 2 page tables. In this case, we expect more valid non-null entries. We continue with the same method. Each Level 2 page table entry points to the starting address of a Level 3 page table, and finally each Level 3 page table entry points to a Level 4 page table. We expect more and more valid entries at each level. Finally, an entry in a Level 4 page table stores the corresponding frame's address. The process of address translation is thus complete. Each entry of the Level 4 page table is known as a *page table entry*. We shall later see that it contains the physical address of the frame and contains a few more important pieces of information.

Note that we had to go through 4 levels to translate a virtual address to a physical address. Reading the page table is thus a slow operation. If parts of this table are in the caches, then the operation may be faster. However, in the worst case, we need to make 4 reads to main memory, which requires more than 1000 cycles. This is a very slow operation. The sad part is that we need to do this for every memory access !!! This is clearly an infeasible idea given that roughly a third of the instructions are memory accesses.

The TLB (Translation Lookaside Buffer)

The notion of spending approximately 1000 cycles for translating a memory address is absolutely impractical. It turns out that we can use the same old notions of temporal and spatial locality to propose a very efficient solution that almost totally hides the address translation latency.

Observe that a page is actually a lot of memory. If we consider a 4 KB page, it can hold 1024 integers. If we assume some degree of locality – temporal and spatial – then there is a high chance that most of the accesses in a short period of time will fall within the range of a few pages. The same observation holds true for code pages as well. There is much more locality at the page level than at the cache block level primarily because of the larger size of a page. If we create a small and fast hardware cache that stores the mappings that have been used recently, it should have a very high hit rate. In fact, it has been shown that caching just 64 to 128 entries is good enough. The hit rate can be as high as 99%.

Hence, almost all processors have a small hardware cache called a TLB (Translation Lookaside Buffer). It caches a few hundred frequently used virtual to physical mappings. Modern TLBs have two levels (L1 TLB and L2 TLB) and cache roughly 1000 entries. We can also have two different TLBs per core: one for instructions and one for data. Given that the L1 TLB is quite small, it can be accessed very quickly – typically in less than a cycle. In a large out-of-order pipeline on a modern core, this small latency is hardly perceptible and as a result, the address translation basically happens for free.

In the rare case, when there is a miss in the TLB, it is necessary to access the page table, which is a slow process. It can take hundreds to thousands of cycles to access the page table. Subsequently, it is necessary to add the virtual→physical mapping to the TLB. Without adding a mapping to the TLB,

it cannot be used to translate addresses.

> **Trivia 2.2.2**
>
> The page table is a software data structure. It is not implemented using custom hardware. Whenever, there is a TLB miss, it is necessary to traverse or *walk* the page table and find the frame corresponding to the page. Walking the page table can be done in software by an OS process, or it can be done by a dedicated hardware module. High-performance Intel and AMD processors have hardware page walkers. Note that it is necessary to add the mapping to the TLB first, otherwise it cannot be used.

Solution to the Size Problem: Swap Space

Let us now solve the size problem. The problem is to run a program with a large memory footprint on a machine with inadequate physical memory. The solution is quite simple. We reserve a contiguous chunk of bytes on a storage device such as the hard disk or a flash drive and use it to extend the physical address space (set of all addressable physical locations). This reserved region is known as the *swap space*. In theory, the storage device need not be connected to the same machine. It can be on a friend's machine and can be accessed over the network.

Whenever we need more space than what physical memory can provide, we take up space in the swap space. Frames can be resident either in physical memory or in the swap space. However, for them to be usable, they need to be brought into main memory first. They cannot directly be used in the swap space.

Let us now go over the entire process. The processor computes the virtual address based on the program logic. This address is translated to a physical address using the TLB. If a valid translation or mapping exists, then the physical address is sent to the memory system: instruction cache or L1 data cache. The access traverses the memory system until the corresponding cache block is found. Even if it is not found in all the caches, it is guaranteed to be present in main memory. However, in the rare case when an entry is not there in the TLB, we record a *TLB miss*. There is a need to walk the page table, which is a slow process.

If the page table has a valid translation (frame present in main memory), then there is a need to first bring this mapping into the TLB. Note that most modern processors cannot use the mapping directly. They need to add it to the TLB first, and then <u>reissue</u> the memory instruction. The second time around, the mapping will be found in the TLB. When a new mapping is added to the TLB, a need may arise to evict an earlier entry. An LRU scheme can be followed to realize this.

If a mapping is not found in the page table, then we can have several situations. The first is that the address is illegal. Then an exception needs to be raised. There might be a need to terminate the program in this case.

However, it is possible that the entry indicates that the frame is not in memory, it is in the swap space. This situation is known as a *page fault*. There is a need to bring the frame from the swap space to the main memory. If the

main memory is full, then we need to evict a frame by writing it to a location on the swap space. Its corresponding page table entry and TLB entry need to be updated. Let us elaborate.

We can store a single bit in a page table entry that indicates if the frame is in the main memory or in the swap space. In the latter case, we will have a page fault. For example, a value of 1 may indicate that the frame is in main memory and 0 may indicate that the frame is in the swap space. In theory, it is possible that we do not have a single swap space. We rather have several swap spaces hosted on multiple storage devices. Hence, a page table entry can be more expressive. It can store the location of the frame and the location of the swap space: id of the device that contains it and a unique device-specific identifier. The device itself can have a complex description that could be a combination of an IP address and a device id. All of this information needs to be stored in the page table entry. This will allow us to host the swap space on any local or remote storage device.

A page fault will involve reading the frame from the swap space into main memory, and then updating the corresponding TLB and page table entries. In this process, if a frame is evicted from main memory to create space, then its entries need to be updated as well.

Permission Bits in a Page Table Entry

The page tables and TLBs store some additional information. They store some permission information. For security reasons, a program is typically not allowed to write to *code* pages. Otherwise, it is easy for viruses to modify the code pages such that a program can execute code that an attacker wants it to execute. Sometimes, we want to create an execute-only page if there are specific licensing requirements where the developers don't want user programs to read the code that is being executed. This makes it easy to find loopholes. We can thus associate three permission bits with each page: read, write and execute. If a bit is 1, then the corresponding permission is granted to the page. For instance, if the bits are 101, then it means that the user process can read and execute code in the page, but it cannot write to the page. These bits are stored in each page table entry and also in each TLB entry. The core needs to ensure that the permission bits are respected.

We can additionally have a bit that indicates whether the page can be accessed by the kernel or not. Most OS kernel pages are not accessible in user space. The page table can store this protection information. This stops user pages from accessing and mapping kernel pages.

Sometimes, the page is present in memory, but the user process does not have adequate permissions to access the page. This is known as a *soft page fault*, which usually generates an exception. When we discuss the MGLRU page replacement algorithm, we shall observe that sometimes this mechanism proves to be quite handy. We can deliberately induce soft page faults to track page accesses. This gives us an idea about the popularity of a process's pages.

> **Definition 2.2.7** Page Faults
>
> A *page fault* is an exception condition, where a mapping is present in the page table, but the corresponding frame is not present in main memory. There is a need for the OS to transfer the frame from the swap space to main memory. This is a very slow and time-consuming process.
>
> Sometimes the page is present in memory, but the user process does not have adequate access permissions. This is a *soft page fault*. In general, such accesses are denied. However, there are other use cases also. Popular page replacement algorithms like the MGLRU algorithm deliberately induce soft page faults to track page accesses. They subsequently change the permission bits and allow the accesses to go through.

Shared Memory Channel

It is true that in the general case, we would like to have the virtual address spaces of two processes separate. This is for correctness and security. However, there are instances when we want two virtual pages in two different processes to actually map to the same frame. Note that this process needs to be highly regulated, and it should happen with the consent of both the processes and the OS. However, if such a mapping can be established safely, then it is very beneficial. We can use it as a shared memory data transfer channel between the two processes. They can transfer data to each other very quickly without the involvement of the OS. Linux's standard C library supports the `shmget` and `shmat` functions for creating and attaching shared memory segments, respectively.

Inverted Page Table

There is often a need to do the reverse – for a physical frame find all the virtual pages across processes that map to it. There are several interesting use cases for such a structure, which is known as an inverted page table. This process is known as *reverse mapping*. Assume that a frame needs to be displaced from main memory because it was chosen by the page replacement algorithm. In this case, the page tables of all the processes that store a mapping to it need to be updated. This means that we need to initiate a process of reverse mapping, which yields a list of pages that map to this frame. The page table entries of each of those pages need to be updated. The same holds true if we just want to change the permission bits of the frame. We shall see in the next chapter that this information is needed while creating a clone of a process (`fork` and `clone` system calls). In secure systems such as Intel SGX® (Secure Guard Extensions) and Intel TDX® (Trust Domain Extensions), it is necessary to maintain an inverted page table such that secure pages are not mapped to unsecure processes.

> **Definition 2.2.8** Inverted Page Table
>
> An inverted page table maps a physical frame to all the virtual pages (across processes) that are mapped to it.

We shall discuss elaborate reverse mapping mechanisms in Section 6.3.1. In general the idea is to have an entry in a table for each physical frame or a group of physical frames. Given that the number of physical frames is much smaller than the number of virtual pages, and they are not process-specific, this can be done. Each entry of this table points to a list of virtual pages. We shall observe in Section 6.3.1 that we can reduce the space requirements of such structures by trading off time with space.

2.2.4 Segmented Memory

Let us now add one more virtualization layer and create "virtualized virtual memory". The x86 architecture has a set of segment registers that add a layer of abstraction on top of virtual memory. The CPU generates logical addresses, which are then converted to linear addresses. The linear addresses are akin to virtual addresses, which are translated to physical addresses. Note that x86 architectures have such a mechanism known as segmented memory. ARM and RISC-V do not have it.

A memory address generated by the CPU is a logical address. This address is added to the contents of a *segment* register. In this case, we are effectively adding an offset to obtain the virtual address. Let us alternatively refer to this as a *linear address* such that it is clearly understood that the contents of a segment register have been added to the CPU-generated logical address. This address is then translated to a physical address using the regular address translation mechanism. Hence, the linear address acts like a virtual address in this case. Some prominent segment registers are the code, data and stack segment registers.

Let us understand the advantages that we gain from segmentation. First, there are historical reasons. There used to be a time when the code and data used to be stored on separate devices. Those days, there was no virtual memory. The physical address space was split between the devices. Furthermore, a base-limit system was used. The segment registers were the base registers. When a program ran, the base register for the code section was the code segment register (`cs` register). Similarly, for data and stack variables, the data and stack segment registers were the base registers, respectively. In this case, different physical addresses were computed based on the contents of the segment registers. The physical addresses sometimes mapped to different physical regions of memory devices or sometimes even different devices.

Given that base-limit addressing has now become obsolete, segment registers have lost their original utility. However, there are new uses. The first and foremost is security. We can prohibit any data access from accessing a code page. This is easy to do. We have seen that in the memory map, the lower addresses are code addresses and once they end, the data region begins. These sections store read-only constants and values stored on the heap. Any data address is a positive offset from the location stored in the data segment register. This means that a data page address will always be greater than any code page address, and thus it is not possible for any data access to modify the regions of memory that store instructions. Most malwares try to access the code section and change instructions such that they can hijack the program and make it do what they want. Segmented addressing is an easy way of preventing such attacks.

In a few other attacks, it is assumed that the addresses of variables and functions in the virtual address space are known. For example, these type of attacks try to modify return addresses stored on the stack. Thus, they need to know the memory address at which the return address is stored. Using the stack segment register, it is possible to obfuscate these addresses and confuse the attacker. In every run, the operating system can randomly set the contents of the stack segment register. This is known as *stack obfuscation.* The attacker will thus not be able to guess what is stored at a given address on the stack – it will change in every run. Note that program correctness is not affected because a program is compiled in a manner where it is assumed that all stack-based addresses are represented as offsets added to the stack pointer.

There are some other ingenious uses as well. Kernel threads often need to quickly access some information such as the id of the previously running user process and a subset of its context. This information needs to be accessed frequently and quickly. The default mechanism is slow. The starting address of this region needs to be loaded into a register. Loading a 64-bit value into a register requires several instructions. This slows down such accesses significantly. We need to access such information using preferably a single instruction. This is only possible if the base address of this region is stored in advance somewhere. The best way to do is to store it in an unused segment register. All accesses to this region can then use this segment register as the base address. Recall that a memory operand can take the segment register as input (see Appendix A).

Segmented Addressing in x86

Figure 2.13 shows the segment registers in the x86 ISA. Some segment registers such as the code segment register can only be modified by privileged software; however, some other segment registers can be modified by user-level processes. There are six such registers per core. Needless to say, they are a part of a process's context. Whenever a new process is loaded, the values in the segment registers also need to be loaded. We have already discussed the code segment register (`cs`), the data segment register (`ds`) and the stack segment register (`ss`). There are three additional segment registers for custom segments that the OS can define. They are `es`, `fs` and `gs`. They are used to store information about the previously running thread, as we shall see in the next chapter.
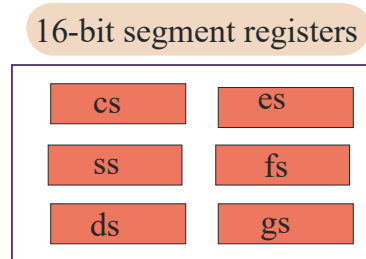
Figure 2.13: The segment registers in the x86 ISA

Figure 2.14 shows the way that segmented memory is addressed. The philosophy is broadly similar to the paging system for virtual memory where the

insight is to leverage temporal and spatial locality as much as possible. There is a segment descriptor cache (SDC) that caches segment descriptors. A segment descriptor contains the base address of the segment, a limit on the segment's size, type, execute permission and privilege level. Most of the time, the value of the corresponding segment descriptor is found in the SDC. Segment descriptors get updated far more infrequently as compared to TLB values. They are only updated on a context switch.

Similar to a TLB miss, if there is a miss in the SDC, there is a need to search in a larger structure. In the older days, there used to be an LDT (local descriptor table) and a global descriptor table (GDT). We can think of the LDT as the L1 level (per process) and the GDT as the L2 level. However, the LDT stopped being used, when there was a transition to 32-bit x86. Nowadays, if there is a miss in the SDC, then a dedicated piece of hardware searches for the value in the GDT, which is a hardware structure. It has a limit of 8191 entries (13-bit addressing and one entry reserved). If there is a miss in the GDT, an interrupt is raised. The operating system needs to populate the GDT with the correct value as it does for the TLB. After a transition to x86-64, the segment registers stopped being used altogether other than the `fs` and `gs` registers. These two registers are still used by the kernel to point to specific memory regions in the kernel's address space. They obviate the need for loading 64-bit addresses into registers. Instead, they are readily available in segment registers.
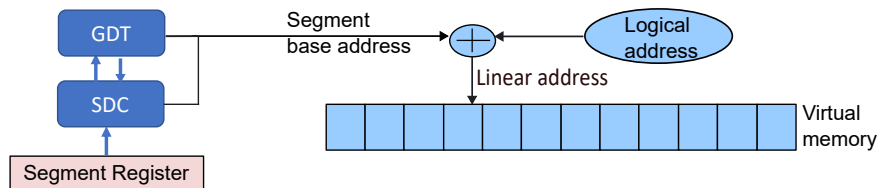
Figure 2.14: Computing the virtual address with memory segmentation

As seen in Figure 2.14, the base address stored in the relevant segment register is added to the logical address. The resultant linear address further undergoes translation to generate the physical address. This is then sent to the memory system.

2.3 I/O System

Any computing machine needs to read inputs from the user and needs to relay the output back to the user. In other words, there needs to be a method to interact with the machine. Hence, we require an input/output or I/O system where programs running on the CPU can interact with devices such as the mouse, keyboard and monitor. These devices could be sending data to the CPU or receiving data from it. Bidirectional transfer is also possible, such as communication with a network device.

It should be easy for a user process to interact with the I/O devices. It is the job of the operating system to provide an interface that is elegant, convenient, safe and fast. Along with software support in the OS, we shall see that we also need to add a fair amount of hardware on the motherboard to ensure that

the I/O devices are properly interfaced. These additional chips comprise the chipset. The motherboard is the printed circuit board that houses the CPUs, memory chips, the chipset and the I/O interfacing hardware ports.

2.3.1 Overview
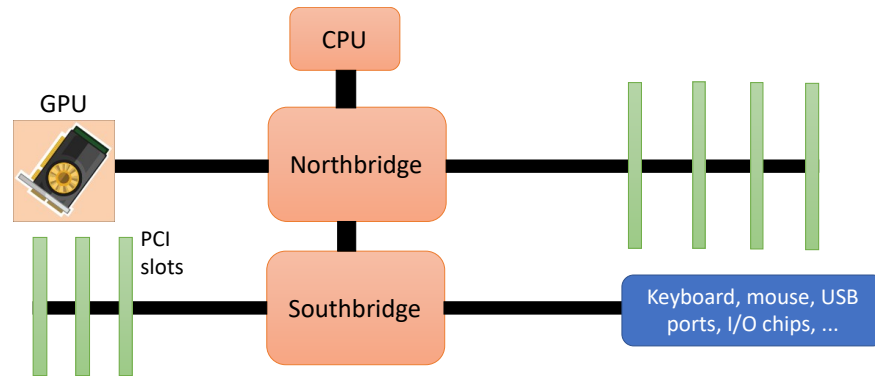
Figure 2.15: Overview of the I/O system

Any processor chip has hundreds of pins. Complex designs have roughly a 1000+ pins. Most of them are there to supply current to the chip: power and ground pins. We need so many pins because modern processors draw a lot of current. Note that a pin has limited current delivery capacity. However, a few hundred pins are typically left for communication with external entities such as the memory chips, off-chip GPUs and I/O devices.

Memory chips have their dedicated memory controllers on-chip. These memory controllers are aware of the number of memory chips that are connected and how to interact with them. This happens at the hardware level and the OS is blissfully unaware of what goes on here. Depending on the motherboard, there could be a dedicated connection to an off-chip GPU. An ultra-fast and high-bandwidth connection is required to a GPU that is housed separately on the motherboard. Such buses (sets of copper wires) have their own controllers that are typically on-chip.

Figure 2.15 shows a traditional design where the dedicated circuitry for communicating with the main memory modules and the GPU are combined, and added to the Northbridge chip. The Northbridge chip used to traditionally be resident on the motherboard (outside the chip). However, in most modern processors today, the logic used in the Northbridge chip has moved into the main CPU chip. It is much faster for the cores and caches to communicate with an on-chip component. Given that both the main memory and GPU have very high bandwidth requirements, this design decision makes sense. Alternative designs are also possible where the Northbridge logic is split into two and is placed at different ends of the chip: one part communicates with the GPU and the other part communicates with the memory modules.

To communicate with other slower I/O devices such as the keyboard, mouse, USB devices and the hard disk, a dedicated controller chip called the Southbridge chip is used. In most modern designs, this chip is resident outside the

chip – it is placed on the motherboard. Typically, there is a bus that connects the Northbridge and Southbridge chips. However, this is not mandatory. There could be a separate connection to the Southbridge chip and in a high-performance implementation, we can have the Southbridge logic inside the CPU chip also. Let us however stick to the simplistic design shown in Figure 2.15.

The Southbridge chip is further connected to dedicated chips in the chipset whose job is to route messages to the large number of I/O devices that are present in a typical system. In fact, we can have a tree of such chips, where messages are progressively routed to the I/O devices through the different levels of the tree. For example, the Soutbridge chip may send messages to the PCI-X chip (PCI eXpress), which needs to subsequently send them to connected I/O devices. The Southbridge chip may also choose to send a message to the USB ports. A router in the chipset routes the message to the destination USB port.

The question that we need to answer is how do we programmatically interact with these I/O ports? It should be possible for assembly programs to read and write from I/O ports easily. There are several methods in modern processors. There is a trade-off between the ease of programming, latency and achievable bandwidth.

2.3.2 Port-Mapped I/O

The simplest method is to use the `in` and `out` instructions in the x86 ISA. They use the notion of an *I/O port* for all their I/O. Let us elaborate. All the connected devices and their associated controllers expose themselves as I/O ports to software (read executables and assembly programs). An I/O port in this case is different from the hardware ports that we find on the side of a laptop like the USB ports. An I/O port in this case is an address in the I/O address space.

In an x86 system, there are typically 2^{16} (64k) 1-byte I/O ports. This is the I/O address space of the system. Each device is assigned a set of I/O ports during boot time. The hardware on the chipset ensures that reads and writes to the I/O ports are relayed to the underlying device. When the operating system boots, it becomes aware of the devices that are connected to the system and the I/O ports that they are mapped to. This is one of the first post-boot tasks that the operating system executes. This information is made available to device drivers – specialized programs that within the OS that communicate with I/O devices. The controllers in the chipset know how to route messages between the CPU and I/O ports.

The size of an I/O port is 1 byte. However, it is possible to address a set of contiguous I/O ports together and read/write 2 or 4 bytes at once. It is important to note that a 2-byte access actually reads/writes two consecutive I/O ports, and a 4-byte access reads/writes four consecutive I/O ports. There are I/O controller chips in the chipset such as the Northbridge and Southbridge chips that know the locations of the I/O ports on the motherboard and can route the traffic to/from the CPUs.

The device drivers incorporate assembly code that uses variants of the `in` and `out` instructions to access I/O ports corresponding to the devices. User-level programs request the operating system for I/O services where they request the OS to effect a read or write. The OS in turn passes on the request to the device drivers, who use a series of I/O instructions to interact with the devices.

Once, the read/write operation is done the data read from the device and the status of the operation is passed on to the program that requested for the I/O operation.

If we dive in further, we observe that an `in` instruction is a message that is sent to the chip on the motherboard that is directly connected to the I/O device. Its job is to further interpret this instruction and send device-level commands to the device. It is expected that the chip on the motherboard knows which message needs to be sent. The OS need not concern itself with such low-level details. For example, a small chip on the motherboard knows how to interact with USB devices. It handles all the I/O. It just exposes a set of I/O ports to the CPU that are accessible via the *in/out* ports. Similar is the case for `out` instructions, where the device drivers simply write data to I/O ports. The corresponding chip on the motherboard knows how to translate this to device-level commands.

Using I/O ports is the oldest method to realize I/O operations and has been around for the last fifty years. It is however a very slow method and the amount of data that can be transferred is very little. Also, for transferring a small amount of data (1-4 bytes), there is a need to issue a new I/O instruction. This method is alright for control messages but not for data messages in high bandwidth devices like the network cards. There is a need for a faster method. This is known as port-mapped I/O (PMIO).

2.3.3   Memory-Mapped I/O
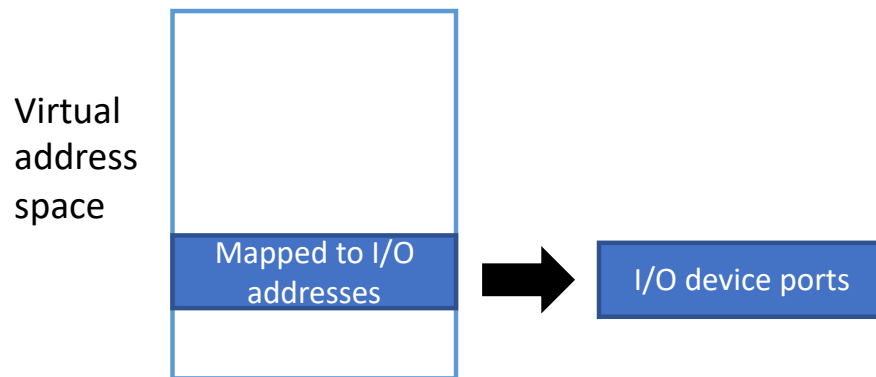
Figure 2.16: Memory-mapped I/O

The faster method is to directly map regions of the virtual address space to an I/O device. Insofar as the OS is concerned, it makes regular reads and writes. The TLB however stores an additional bit indicating that the page is an I/O page. The hardware automatically translates memory requests to I/O requests. There are several advantages of this scheme (refer to Figure 2.16).

The first is that we can send a large amount of data in one go. The x86 architecture has instructions such as `rep movs` and `rep stos` that enable the programmer to move hundreds of bytes between addresses in one go. These instructions can be used to transfer kilobytes to/from I/O space. The hardware

on the chipset can then use fast mechanisms to ensure that this process is realized as soon as possible.

At the side of the processor, we can clearly see the advantage. All that we need is a few instructions to transfer a large amount of data. This reduces the instruction processing overhead at the end of the CPU and keeps the program simple – we only need to use load and store instructions. I/O devices and chips in the chipset have also evolved to support memory-mapped I/O. Along with their traditional port-based interface, they are also incorporating small memories that are accessible to chips in the chipset. The data that is in the process of being transferred to/from I/O devices can be temporarily buffered in these small memories.

A combination of these technologies makes memory-mapped I/O very efficient. Hence, it is very popular as of 2025. In many reference manuals, it is conveniently referred to by its acronym *MMIO*.

2.3.4   DMA
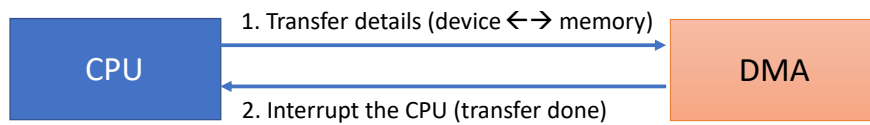
Figure 2.17: I/O using DMA

Even though memory-mapped I/O is much more efficient than the older method that relied on primitive instructions and basic I/O ports, it turns out that we can do far better. Even in the case of memory-mapped I/O, the processor needs to wait for the load/store instruction that is doing the I/O to finish. Given that I/O operations take a lot of time, the entire pipeline fills up and the processor remains stalled until the outstanding I/O operations complete. One simple solution is that we do the memory-mapped I/O operations in small chunks and do other work in the middle; however, this slows down the entire transfer process. We can also remove write operations from the critical path and assume that they are done asynchronously. Still the problem of slow reads will be there.

Our main objective here is that we would like to do other work while I/O operations are in progress. We can extend the idea of asynchronous writes to also have asynchronous reads. In this model, the processor does not wait for the read or write operation to complete. The key idea is shown in Figure 2.17, where there is a separate DMA (direct memory access) chip that effects the transfers between the I/O device and memory. The CPU basically <u>outsources</u> the I/O operation to the DMA chip. The chip is provided with the addresses in memory as well as the addresses on the I/O device along with the direction of data transfer. Subsequently, the DMA chip initiates the process of data transfer. In the meanwhile, the CPU can continue executing programs without stalling. Once the DMA operation completes, it is necessary to let the OS know about it.

Hence, the DMA chip issues an interrupt, the OS comes into play, and then it realizes that the DMA operation has completed. Since user programs cannot

directly issue DMA requests, they instead just make system calls and let the OS know about their intent to access an I/O device. This interface can be kept simple primarily because it is only the OS's device drivers that interact with the DMA chip.

When the interrupt from the DMA controller arrives, the OS knows what to do with it and how to signal the device drivers that the I/O operation is done. The device driver can then either read the data that has been fetched from an I/O device or assume that the write has completed. In many cases, it is important to let the user program also know that the I/O operation has completed. For example, when the printer successfully finishes printing a page, the icon changes from "printing in progress" to "printing complete". Signals can be used for this purpose.

To summarize, in this section we have seen three different approaches for interacting with I/O devices. The first approach is also the oldest approach where we use old-fashioned I/O ports. This is a simple approach especially when we are performing extremely low-level accesses, and we are not reading or writing a lot of data. Currently, I/O ports are primarily used for interacting with the BIOS (booting system), simple devices like LEDs and in embedded systems. This method has mostly been replaced by memory-mapped I/O (MMIO). MMIO is easy for programmers, and it leverages the natural strengths of the virtual memory system. It provides a convenient and elegant interface for device drivers – they use regular load/store instructions to perform I/O. Also, another advantage is that it is possible to implement a zero-copy mechanism where if some data is read from an I/O device, it is very easy to transfer it to a user program. The device driver can simply change the mapping of the pages and map them to the user program after the I/O device has populated the pages. Consequently, there is no necessity to read data from an I/O device into pages that are accessible only to the OS, and then copy all the data once again to user pages. This is inefficient.

Subsequently, we looked at a method, which provides much more bandwidth and also does not stall the CPU. This is known as DMA (direct memory access). Here, the entire role of interacting with I/O devices is outsourced to an off-chip DMA device; it finally interrupts the CPU once the I/O operation completes. After that the device driver can take appropriate action, which also includes letting the user program know that its I/O operation is over.

Point 2.3.1

There are three kinds of methods to perform I/O operations from a software perspective.

PMIO We rely on classical I/O ports, which are small 1-4-byte I/O locations in the system's I/O address space (total size: 64 KB). The `in` and `out` x86 instructions can be used to read and write bytes from I/O ports, respectively.

MMIO To enable faster transfers, a part of the virtual address space of a process can be mapped to an I/O device's internal memory. I/O operations can now be realized with simple load and store instructions. x86 has specialized instructions that can transfer large chunks of data in one go.

> **DMA** The entire job of effecting the transfer is outsourced to the DMA chip (or DMA controller). After performing the transfer, it raises an interrupt to let the OS know that the transfer has completed.

2.4   Summary and Further Reading

2.4.1   Summary

Summary 2.4.1

1. Each core has general-purpose registers and privileged registers. The latter are used by the operating system or virtual machine monitor. They can be used to control the behavior of hardware at a very low-level.

2. Modern processors define three privilege rings. The OS kernel operates at ring 0, and the application operates in ring 3. Middleware software operate in rings 1 and 2.

3. Kernel processes do not run all the time. They can be invoked via only three mechanisms: hardware interrupts, program-generate exceptions and system calls.

4. The kernel uses signals to communicate with user processes.

5. The values stored in the general purpose registers, a few privileged registers and the program counter comprise the *context* of a process. The values in memory remain intact during context switches. Hence, they need not be a part of the context that is saved and restored upon every context switch.

6. There is a need to have a timer chip that generates a timer interrupt periodically. It is a guaranteed source of interrupts. It is needed to ensure that kernel processes run periodically and perform tasks related to scheduling, memory and device management.

7. If there is a need to run a kernel process on a remote core, then an IPI (inter-processor interrupt) needs to be sent to it. The remote core's interrupt handler will invoke the relevant interrupt handler.

8. Whenever the core is notified of an event of interest (exception, system call, interrupt), it pauses the currently running process, stores its context, runs the interrupt handler, finishes other kernel work, possibly schedules other user processes and finally restores the context of the process that was paused.

9. A process assumes that it owns a large contiguous memory space. For example, if it runs on an n-bit machine, it may assume that it can access any location from 0 to $2^n - 1$ at will. Moreover, all

processes lay out their code, data, heap and stack sections in a similar manner in memory. This is known as the memory map of the process. The stack typically starts at a very high address and grows downwards.

10. This "virtual view" of memory is an elegant and convenient abstraction for the compiler, programmer and processor. However, in a practical real-world system, there are three problems.

    Compatibility Problem There is a need to translate 48 or 64-bit addresses in the assumed conceptual address space to physical addresses that are in a much smaller address space. The size of the physical address space is basically equal to the number of available physical locations (in the main memory and storage devices).

    Overlap Problem Two different processes should never access the same set of addresses, unless otherwise intended.

    Size Problem We would like to run programs whose memory footprint is greater than the size of the main memory.

11. The virtual memory subsystem solves all of these problems. It divides the virtual address space (assumed by the process) into 4-KB pages. It does the same with the physical address space and divides it into many 4-KB frames. A dedicated multi-level page table maintains the mapping between a virtual page and a physical frame. It never maps two different virtual pages to a physical frame unless this is the original intention.

12. To speed up the translation process a small hardware cache containing frequently-used mappings is used. It is known as the Translation Lookaside Buffer (TLB).

13. To augment the size of the physical address space, some space in storage devices can be used. This is known as the swap space. If a frame is not found in main memory, then this event is known as a page fault. There is a need to bring in the frame from swap space and possibly replace a frame already resident in main memory.

14. x86 uses six different segment registers. The CPU generates a logical address that is added to the base address stored in the associated segment descriptor. The resultant linear address acts like a virtual address that is translated to a physical address.

15. x86-64 primarily uses the `fs` and `gs` segments.

16. All these segment registers contain an index that maps to a segment descriptor in the GDT table. The lookup process is accelerated by using a segment descriptor cache.

17. There are three methods for performing I/O: port-mapped I/O (using regular I/O ports, 64-KB I/O address space), memory-mapped I/O (map a region of the virtual address space to the I/O device's internal memory) and DMA (outsource the job of transferring data to a dedicated off-chip circuit).

2.4.2 Further Reading

This chapter has discussed a lot of computer architecture concepts. For further reading, the best resources are standard textbooks in computer architecture such as the books written by your author: Basic Computer Architecture [Sarangi, 2021] and Next-Gen Computer Architecture [Sarangi, 2023]. They will provide the reader with a thorough understanding of all the architectural mechanisms that are used in modern processors.

For readers who are interested in how privileged instructions operate and control low-level aspects of hardware, the most comprehensive resources are Intel's software developer manuals. Volume 3 [Corporation, 2024a] is a guide for system programmers. This manual describes paging, protection, interrupt handling, memory management and debugging in great detail. All OS developers must read this manual. Volume 4 [Corporation, 2024b] describes all model-specific registers (MSRs). Recall that all low-level hardware functions can be programmed and controlled by writing values to MSRs.

Exercises

Ex. 1 — Why are multiple rings there in an x86 processor? Isn't having just two rings enough?

Ex. 2 — How does a process know that it is time for another process to run in a multitasking system? Explain the mechanism in detail.

Ex. 3 — Assume a 16-core system. There are 25 active threads that are purely computational. They do not make system calls. The I/O activity in the system is negligible. Answer the following questions:

a) How will the scheduler get invoked?

b) Assume that the scheduler has a special feature. Whenever it is invoked, it will schedule a new thread on the core on which it was invoked and replace the thread running on a different core with another active (ready to run) thread. How do we achieve this? What kind of hardware support is required?

Ex. 4 — What is the need for having privileged registers in a system? How does Intel avoid them to a large extent?

Ex. 5 — How can we design a virtual memory system for a machine that does not have any kind of storage device such as a hard disk attached to it? How do we boot such a system?

Ex. 6 — Assume a system has practically infinite amount of physical memory (much more than what all the processes need). Would we still need virtual memory? Justify your answer.

Ex. 7 — Do the processor and compiler work with physical addresses or virtual addresses?

Ex. 8 — How does the memory map of a process influence the design of a page table for 64-bit systems?

Ex. 9 — What are the advantages of segment-based memory addressing?

Ex. 10 — Can segments be used in place of privileged registers in the context switch process? Can we create storage space akin to a scratch pad (temporary storage area)?

Ex. 11 — How does segmentation allow us to define per-CPU memory regions? Where are these regions (possibly) stored in the virtual address space? Why is this more efficient than other methods that rely on storing data at pre-specified locations on the stack?

Ex. 12 — When is it preferred to use an inverted page table over the traditional (tree-based) page table?

Ex. 13 — Why are the memory contents not a part of a process's context?

Ex. 14 — Assume two processes access a file in read-only mode. They use memory-mapped I/O. Is there a possibility of saving physical memory space here?

Chapter 3

Processes

The concept of a *process* is arguably the most important concept in operating systems. A process is simply defined as a program in execution. A program or an executable is represented as a file in the ELF format (refer to Appendix B) on the disk. Within the file system[1], it lies dormant – it does not execute. A program is brought to life when the user invokes a command to run the program and the loader loads the code and data into memory. Subsequently, it sets the program counter to the starting address of the first instruction in the text section of the memory image of the newly created process. This process then begins to execute. It can have a long life and can make system calls, receive messages from the OS in terms of signals, and the OS can swap the process in and out a countless number of times. A process may undergo thousands of context switches before it is finally destroyed.

A process during its execution can acquire a lot of resources. It finally releases them once it terminates. Some resources include a right to use the CPU for some time, memory space, open file and network connections. Processes have a very elaborate interface for interacting with the OS. Bidirectional communication is possible. As we have discussed earlier, there are two popular mechanisms: system calls are used to request services from the OS, whereas signals are used by the OS to communicate information back to a process.

We shall learn in this section that creating data structures to represent information about a process is the key challenge. Most of the algorithms that are subsequently used are straightforward and taught in a regular course on algorithms. However, using the right combination of data structures and representing information in a way that it is quickly accessible without unnecessary redundancy is much more difficult than it sounds. There is a need to design several data structures to represent information, embed and interconnect them such that there is very little redundancy and the diverse nature of a process is nicely captured.

We shall also learn that the process of creating and destroying a process is quite tricky. Linux follows the same mechanisms that other Unix-like operating

---

[1]The file system is a hierarchical organization of files on a storage device. A file is defined as a logically contiguous sequence of bytes on a storage device. Files can be of various types such as documents, video files, audio files, and so on.

systems use. They first fully copy the parent process's memory image and then replace the memory image if there is a need. This approach is however not followed in other operating systems like Windows. We shall learn more about Linux's approach and its pros and cons. Finally, we shall also spend some time in understanding the different kinds of context switch mechanisms that are needed in a modern operating system. Some of them can be made more efficient and admit optimizations.

Organization of this Chapter
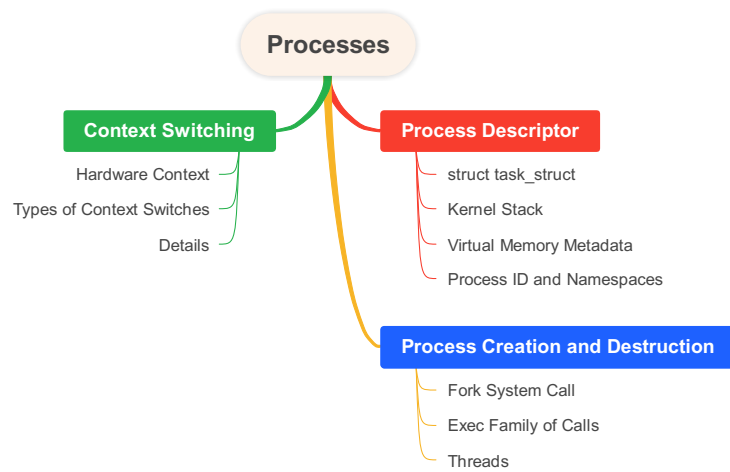
Figure 3.1: Organization of this chapter

This chapter has three subparts (refer to Figure 3.1). We will start with discussing the main concepts underlying a process in the latest Linux kernel. A process is a very complex entity because the kernel needs to create several data structures to represent all the runtime state of the running program. This would, for example, include creating elaborate data structures to manage all the memory regions that the process owns. This makes it easy to allocate resources to processes and later on deallocate them. The kernel uses the `task_struct` structure to maintain this information.

Subsequently, we shall discuss the relevant code for managing process ids (*pid*s in Linux) and the overall state of the process. We shall specifically look at a data structure called a maple tree, which the current version of the Linux kernel uses extensively. We shall then also look at two more kinds of trees, which are very useful for searching data namely the radix tree and the augmented tree. Appendix C describes these data structures in great detail. It is thus necessary to keep referring to it.

In the subsequent section, we shall look at the methods of process creation and destruction. Specifically, we shall look at the `fork` and `exec` system calls. Using the `fork` system call, we can clone an existing process. Then, we can use the `exec` family of calls to superimpose the image of a different executable on

top of the currently running process. This is the standard mechanism by which new processes are created in Linux.

Finally, we shall discuss the context switch mechanism in a fair amount of detail. We shall first introduce the different types of context switches and the state that the kernel needs to maintain to suspend a running process and resume it later. The process of suspension and resumption of a process is different for different kinds of processes. For instance, if we are running an interrupt handler, then certain rules apply that are quite restrictive whereas if we are running a regular program, then some other rules apply.

Summary: Data Structures used in this Chapter

The reader is requested to kindly take a look at some important data structures that are used in the Linux kernel (see Appendix C). Before proceeding forward, we would like the reader to be fully familiar with the following data structures: B-tree, B+ tree, maple tree and radix tree. They are extensively used throughout the kernel. It is important to understand them before we proceed.

The kernel heavily relies on tree-based data structures. We frequently face problems such as identifying the virtual memory region that contains a given virtual memory address. This boils down to a search problem – given a key find the value. Often using a hash table is not a very wise idea in such cases, particularly when we are not sure of how many keys we need to store. They also have poor cache locality and do not lend themselves to easily implementing range queries. Trees, on the other hand, are very versatile data structures. With logarithmic-time complexity they can implement a wide variety of functions. They support concurrent accesses and highly cache-efficient organizations. This is why they are often used in high-performance implementations. Trees are naturally scalable as well in terms of the number of nodes that they store.

We can always use the classical red-black and AVL trees. However, it is far more common to use m-ary B-trees where a node's size is equal to that of one or more cache blocks. This leads to minimizing cache block fetches and also allows convenient node-level locking. A B+ tree is a variation of a B-tree where the keys are only stored at the leaves. A maple tree is a specialized B+ tree that is used in the Linux kernel. The arity of the nodes changes with the level. Internal nodes close to the root typically have fewer children and nodes with higher depths have more children. Such adaptive node sizing is done to improve memory efficiency.

Another noteworthy structure for storing keys and values is the radix tree. It works well if keys share common prefixes. We traverse such a tree based on the digits in the key. The search time is linear in terms of the number of digits in the key.

The kernel also uses augmented trees that help us solve problems of the following type: given a bit vector, find the location of the first 0 or 1 in logarithmic time (starting from a given location and proceeding towards higher or lower indexes). Such trees are used to accelerate operations on bit vectors especially scan operations that attempt to find the next 0 or 1 in a bit vector. The implementation can be optimized. Each leaf of the augmented tree need not correspond to a single bit. It can instead correspond to a set of 32 or 64 bits (size of a memory word). Their parent node in the tree just needs to store if any of those 32 or 64 bits are equal to a 0 or 1 or not. Note that in modern machines data can only be stored at the granularity of 32 or 64 bits; hence, the

tree needs to be designed in such a manner. Moreover, to speed up accesses, a parent node can store the status of each of its children. It stores whether a child (which is the root of a subtree) contains a 0/1 in the range that it spans or not. It is thus not necessary to access the child nodes.

3.1 The Process Descriptor

3.1.1 The Notion of a Process

It is amply clear by now that a process is an entity that is quite multifaceted and this makes it reasonably difficult to represent. It is true that it is an instance of a running program, however, this simple description does not yield to a simple implementation of its descriptor. We need to create elaborate data structures to store information associated with the process: details of the resources that it uses and owns throughout the system.

Processes can run with different privilege levels. We can have user processes, kernel processes and middleware processes (run at ring levels 1 and 2). Furthermore, processes can be standalone (unrelated to other processes) or they can be part of a group of processes that share resources and memory with each other.

The former type of processes are known as single-threaded processes. When we write a regular C program and launch it, a single-threaded process is created. We can alternatively have a multi-threaded process, which actually represents a group of processes. The individual processes known as *threads* share resources such as a part of the virtual memory space, open files, environment variables and resource limits with each other. A *thread* is thus a *lightweight process*. Specifically, it shares a part of its memory space notably the code, data, bss and heap sections (see Appendix B) with other threads along with resources like open file and network connections. Think of a thread as a process in its own right. It is an independently schedulable entity, yet it shares some resources with other threads. A thread does not share its stack, register state and thread-local storage area with other threads. Threads in a thread group share a single thread group id (`tgid`). A thread group often has a group leader (leader thread), which is typically the thread that spawned the rest of the threads. Its process id (`pid`) is equal the `tgid` of the entire group of threads.

> **Point 3.1.1**
>
> A thread and a standalone process are actually two points in a spectrum of process definitions. A standalone process does not share anything with other processes. However, truly standalone processes are rare because the code pages of dynamically linked libraries are typically shared. In fact, while creating a process it is possible to exactly specify which resources the child process shall share with the parent process that is creating it. There is thus a spectrum of possibilities. A *thread* is an extremity of this spectrum where it shares as much as it can with other threads in its thread group. All the points in this spectrum can be viewed as regular processes that just have different degrees of inter-process sharing of code and data pages.

We shall revisit this definition in Section 3.3.2, where we shall look at how

threading is implemented. In Linux, different threads in a thread group share the complete virtual address space. Each thread still has its dedicated stack and TLS region. This is achieved by assigning every stack and TLS region to a unique thread-specific point in the shared virtual space.

3.1.2   struct task_struct

Let us now describe the process descriptor, which is the key data structure in the operating system for storing all process-related information. Linux traditionally uses the `struct task_struct` data structure for storing all such process-related bookkeeping information. This data structure keeps all of this information in one place. The key components of the `task_struct` data structure are shown in Table 3.1. Linux internally refers to every process as a `task`.

The approach that we shall follow in this section is to go through each field of the `task_struct` structure one by one. We shall see a complex story unfold in front of us. The reader is advised to read the relevant sections of Appendix C on the implementation of linked lists in the Linux kernel, B-trees, B+ trees, maple trees and augmented trees.

Field	Description
`struct thread_info thread_info`	Low-level information
`uint state`	Process state
`void * stack`	Kernel stack
Priorities	prio, static_prio, normal_prio
`struct sched_info sched_info`	Scheduling information
`struct mm_struct *mm, *active_mm`	Pointer to memory information
`pid_t pid`	Process id
`struct task_struct *parent`	Parent process
`struct list_head children, sibling`	Child and sibling processes
Other fields	file system, I/O, synchronization, and debugging fields

Table 3.1: Key fields in `task_struct`

3.1.3   struct thread_info

Overview of Low-Level Data Types and Structures

A low-level data type is either a primitive data type or a C structure that is hardware-aware. Its design is heavily influenced by how it can be efficiently stored and accessed. For example, a structure that is aware of low-level details has its fields arranged in such a way that they are aligned to cache line boundaries. This minimizes false sharing misses.

The `thread_info` structure used to be the heart of the `task_struct` structure in older kernels. However, it is on its way out now. It is a quintessential example of a low-level data structure. We need to understand that high-level data structures such as linked lists and queues are defined at the software level and their connections with the real hardware are at best tenuous. They are usually not concerned with the details of the machine, the memory layout or

other constraints imposed by the memory system. For instance, we typically do not think of word or variable level alignment in cache lines, etc. Of course, highly optimized libraries care about them, but normal programmers typically do not concern themselves with hardware-level details. However, while implementing an operating system, it becomes quite essential to align the fields of the data structure with actual memory words such that they can be accessed very efficiently. For example, if it is known that a given data structure always starts at a 4 KB page boundary, then it becomes very easy to calculate the addresses of the rest of the fields or solve the inverse problem – find the starting point of the data structure in memory given the address of one of its fields. The `thread_info` structure is a classic example of this.

Before looking at the structure of `thread_info`, let us describe the broad philosophy surrounding the definitions of its constituent fields. The Linux kernel is designed to run on a large variety of machines that have very different instruction set architectures – in fact some may be 32-bit architectures and some may be 64-bit architectures. Linux can also run on very small 16-bit machines as well. We thus want most of the kernel code to be independent of the machine type otherwise it will be very difficult to write the code. Hence, there is an `arch` directory in the kernel that stores all the machine-specific code. The job of the code in this directory is to provide an abstract interface to the rest of the kernel code, which is not machine dependent. For instance, we cannot assume that an integer is four bytes on every platform or a long integer is eight bytes on every platform. These things are quite important for implementing an operating system because many a time we are interested in byte-level information. Hence, to be 100% sure, it is a good idea to define all the primitive data types in the `arch` directories.

For example, if we are interested in defining an unsigned 32-bit integer, we should not use the classic `unsigned int` primitive because we never know whether an `int` is 32 bits or not on the architecture on which we are compiling the kernel. Hence, it is a much better idea to define custom data types. For example, they can guarantee that regardless of the architecture, a data type will always be an unsigned integer (32 bits long). Courtesy the C preprocessor, this can easily be done. We can define types such as `u32` and `u64` that correspond to unsigned 32-bit and 64-bit integers, respectively, on all target architectures. It is the job of the architecture-specific module writers to include the right kind of code in the `arch` folder to implement these virtual data types (`u32` and `u64`). Once this is done, the rest of the kernel code can use these data types seamlessly.

Similar abstractions and virtualization mechanisms are required to implement other parts of the boot subsystem, and other low-level services such as memory management and power management. Basically, anything that is architecture-specific needs to be defined in the corresponding subfolder in the `arch` directory and then a generic interface needs to be exposed to the rest of the kernel code. The rest of the kernel code can be blissfully unaware of architectural details.

Description of `thread_info`

Let us now look at the important fields in the `thread_info` structure. Note that throughout the book, we will not list all the fields in a data structure. We will only list the important ones. In some cases, when it is relevant, we will use

the ellipses . . . symbol to indicate that something is omitted, but most of the time for the sake of readability, we will not have any ellipses.

The declaration of `thread_info` is shown in Listing 3.1.

Listing 3.1: The `thread_info` structure.
source : arch/x86/include/asm/thread_info.h#L56

```
struct thread_info {
    /* Flags for the state of the process, system calls and
        thread synchrony (resp.) */
    unsigned long flags;
    unsigned long syscall_work;
    u32 status;

    /* current CPU */
    u32 cpu;
}
```

This structure basically stores the current state of the thread, the state of the executing system call and synchronization-related information. Along with that, it stores another vital piece of information, which is the number of the CPU on which the thread is running or is scheduled to run at a later point in time. We shall see in later sections that finding the id of the current CPU (and the state associated with it) is a very frequent operation and thus there is a pressing need to realize it as efficiently as possible. In this context, `thread_info` provides a somewhat suboptimal implementation. There are faster mechanisms of doing this, which we shall discuss in later sections. It is important to note that the reader needs to figure out whether we are referring to a thread or a process depending upon the context. In most cases, it does not matter because a thread is treated as a process by the kernel. However, given that we allow multiple threads or a thread group to also be referred to as a multi-threaded process (albeit, in limited contexts), the term *thread* will more often be used because it is more accurate. It basically refers to a single program executing as opposed to multiple related programs (threads) executing.

3.1.4   Task States

Let us now look at the process states in Linux. This is shown in Figure 3.2. In the scheduling world, it is common to refer to a single-threaded process or a thread in a multithreaded process as a *task*. A task is the basic unit of scheduling. We shall use the Linux terminology and refer to any thread that has been started as a task. Let us thus look at the states that a task can be in.

Here is the fun part in Linux. A task that is currently running and a ready task that is queued to run in a CPU-specific runqueue have the same state: TASK_RUNNING. There are historical reasons for this as well as there are simple common sense reasons in terms of efficiency. We are basically saying that a task that is ready to run and one that is running have the same state and thus in a certain sense are indistinguishable. This little trick allows us to use the same queue for maintaining all such tasks that are ready to run or are currently running. This simplifies many design decisions and reduces task state updates. Specifically, if there is a context switch, then there is no need to change the status of the task that was swapped out. Of course, someone may argue that
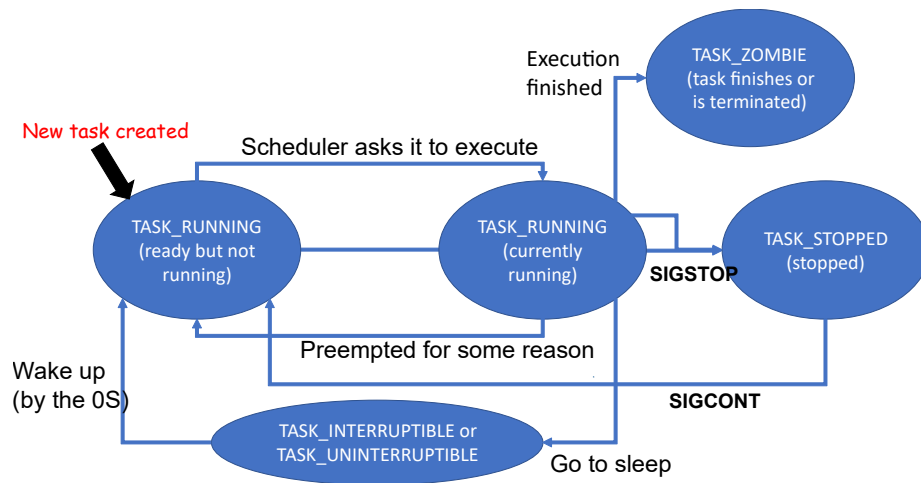
Figure 3.2: Linux task states

using the same state (TASK_RUNNING) introduces ambiguity. To a certain extent it is true, but it does simplify a lot of things and does not appear to be a big hindrance in practice.

Now, it is possible that a running task may keep on running for a long time and the scheduler may decide that it is time to swap it out so that other tasks get a chance. In this case, the task is said to be "preempted". This means that it is forcibly displaced from a core (swapped out). However, it is still ready to run, hence its state remains TASK_RUNNING. Its place is taken by another task – this process thus continues.

Let us look at a few other interactions. A task may be paused by sending it the SIGSTOP signal. Specifically, the kill system call or the command line utility having the same name can be used to send the stop signal to a task. We can also issue the following command on the command line: kill -STOP pid. Another approach is to send the SIGTSTP signal by pressing Ctrl-z on the terminal. The only difference here is that this signal can be ignored. Sometimes there is a need for suspending or pausing a task, especially if we want to run the task at a later point of time when sufficient CPU and memory resources are available. In this case, we can just pause the task. Note that SIGSTOP is a special type of signal that cannot simply be discarded or caught by the process that corresponds to this task. In this case, this is more of a message to the kernel to actually pause the task. It has a very high priority. At a later point of time, the task can be resumed using the SIGCONT signal. Needless to say, the task resumes at the same point at which it was paused. The correctness of the process is not affected unless it relies on some aspect of the environment that possibly got changed while it was in a paused state. The fg command line utility can be used to resume such a suspended task.

Let us now come to the two interrupted states namely INTERRUPTIBLE and UNINTERRUPTIBLE. A task enters these states when it requests for some service like accessing an I/O device, which is expected to take a lot of time. In the first state, INTERRUPTIBLE, the task can still be resumed to act on a

message sent by the OS, which we refer to as a signal. For instance, it is possible for other tasks to send the interrupted process a message (via the OS) and in response it can invoke a signal handler. Recall that a signal handler is a specific function defined in the program that is conceptually similar to an interrupt handler, however, the only difference is that it is implemented in user space. In comparison, in the UNINTERRUPTIBLE state, the task does not respond to signals.

Zombie Tasks

The process of deleting the state of a task after it exits is quite elaborate in Linux. To start with, note that the processor has no way of knowing when a task has completed. It will continue to fetch bytes from memory and try to execute them. It is thus necessary to explicitly inform the kernel that a task has completed by making the `exit` system call. However, a task's state is not cleaned up at this stage. Instead, the task's parent is informed using the `SIGCHLD` signal. Every task has a parent. It is the task that has spawned the current task. The parent then needs to call the system call `wait` to read the exit status of the child. It is important to understand that every time the `exit` system call is called, the exit status is passed as an argument. Typically, the value zero indicates that the task completed successfully. On the other hand, a non-zero status indicates that there was an error. The status in this case represents the error code.

Here again, there is a convention. The exit status '1' indicates that there was an error, however it does not provide any additional details. We can refer to this situation as a non-specific error. Given that we have a structured hierarchy of tasks with parent-child relationships, Linux explicitly wants every parent to read the exit status of all its children. Until a parent task has read the exit status of the child, the child remains a *zombie* task – neither dead nor alive. After the parent has read the exit status, all the state associated with the completed child task can be deleted.

3.1.5 Kernel Stack

Let us ask an important question. Where does the kernel store all the information of a running task when there is a context switch? This is where we come to an important concept namely the *kernel stack*. For every running thread in the user space, there is an associated kernel thread that typically remains dormant when the user thread is executing. The kernel thread uses its own stack to execute. Whenever the user thread makes a system call and requests the kernel for a specific service, instead of spawning a new thread, the OS simply runs the kernel thread associated with the user-level thread. Furthermore, we use the kernel stack associated with the kernel thread. This keeps things simple – the kernel stack becomes a natural home for all thread-related state. We will add some nuance to this simple abstraction in later chapters. However, for the time being, let us continue with this.

There are many limitations associated with the kernel stack given that kernel memory management is complex. Unlike a user-level stack, we do not want it to become arbitrarily large. This will cause a lot of memory management problems.

Hence, typically all versions of the Linux kernel have placed strict hard limits on the size of the kernel stack.

The Kernel Stack in Yesteryears
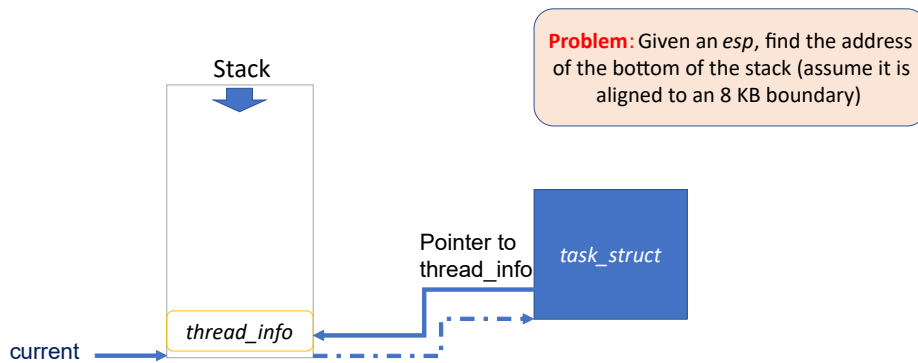
Figure 3.3: The structure of a kernel stack (older versions of Linux)

The size of the kernel stack is limited to two 4-KB pages, i.e., 8 KB. It contains useful data about the running thread. These are basically per-thread stacks. In addition, the kernel maintains a few other stacks, which are specific to a CPU. The CPU-specific stacks are used to run interrupt handlers, for instance. Sometimes, we have very high priority interrupts and some interrupts cannot be ignored (not maskable). The latter kind of interrupts are known as NMIs (non-maskable interrupts). This basically means that if we are executing an interrupt handler, if a higher priority interrupt arrives, we need to do a context switch and run the interrupt handler for the higher-priority interrupt. This is conceptually similar to the regular context switch process for user-level tasks. It is just that in this case interrupt handlers are being paused and subsequently resumed. This is happening within the kernel. Note that each such interrupt handler needs its own stack to execute. Every time an interrupt handler runs, we need to find a free stack and assign it to the handler. Once, the handler finishes running, the stack's contents can be cleared and the stack is ready to be used by another handler. Given that nested interrupts (running interrupt handlers by pausing other handlers) are supported, we need to provision for many stacks. Linux has a limit of 7. This means that the level of interrupt handler nesting is limited to 7.

Figure 3.3 shows the structure of the kernel stack in older kernels. The `thread_info` structure was kept at the lowest address in the 8-KB memory region that stored the kernel stack. Even in current kernels, this memory region is always aligned to an 8-KB boundary. The `thread_info` structure had a variable called `task` that pointed to the corresponding `task_struct` structure. The `current` macro subsumed the logic for getting the `thread_info` of the current task and then getting a pointer to the associated `task_struct` from it. The main aim here is to design a very quick method for retrieving the `task_struct` associated with the current task. This is a very time-critical operation in modern kernels and is invoked very frequently. Hence, a need was felt to optimize this

process as much as possible. Even saving a few instructions provides substantial benefits.

The greatness of this scheme is as follows. From any stack address, we can quickly compute the address at which `thread_info` is stored, which is at the bottom of the 8-KB region. This address is simply the largest multiple of 8 KB that is smaller than the address of a variable on the stack. Simple bitwise operations on the address that involve zeroing out the 13 LSB bits do the trick! Once, we get the address of the `thread_info` structure, we can get the pointer to the `task_struct` with one load operation.

Example 3.1.1

Write a short function in C to find the largest multiple of 8 that is smaller than a given value.

Answer:

```
int find_multiple(int x) {
    return (x & ~7);
}
```

Example 3.1.2

Write a function to extract the ith bit in the number x. The LSB is the first bit.

Answer:

```
int extract(int x, int i){
    return (x & (1 << (i - 1))) >> (i-1);
}
```

Point 3.1.2

Often a need is felt to store a set of bits. Each bit could be a flag or some other status code. The most space-efficient data structure to store such bits is often a variant of the classical unsigned integer. For example, if we want to store 12 bits, it is best to use an unsigned short integer (u16). The 12 LSB bits of the primitive data type can be used to store the 12 bits, respectively. Similarly, if we wish to store 40 bits, it is best to use an unsigned long integer (u64). The bits can be extracted using the logic followed in Example 3.1.2.

The Kernel Stack in the Latest Kernels

The kernel stack as of today looks more or less the same. It is still limited to 8 KB in size. However, the trick involving placing `thread_info` at the lowest address and using that to reference the corresponding `task_struct` is not needed anymore. We can use a better method that relies on segment registers. This is one of the rare instances in which x86 segmentation proves to be extremely

beneficial. It provides a handy reference point in memory for storing specific data that is highly useful and is frequently used. Furthermore, to use segmented addressing, we do not need any extra instructions (see Appendix A). The segment information can be embedded in the memory address itself. Hence, this part comes for free and the Linux kernel designers leverage this to the hilt.

Listing 3.2: The current task
source : arch/x86/include/asm/current.h#L39

```
DECLARE_PER_CPU(struct task_struct *, current_task);

static __always_inline struct task_struct *get_current(void)
{
    return this_cpu_read_stable(current_task);
}
#define current get_current()
```

Refer to the code in Listing 3.2. It defines a macro `current` that returns a pointer to the current `task_struct` via a chain of macros and in-line functions. [2] The code ultimately resolves to a single instruction that reads the address of the current task's `task_struct` in the `gs` segment [Lameter and Kumar, 2014]. The `gs` segment thus serves as a dedicated region that stores information that is quickly accessible to a kernel thread. In fact, the kernel partitions a part of this region to store information specific to each core (CPU in kernel's parlance). It can thus instantly access the `task_struct` structures of processes running on all the CPUs.

Note that here we are using the term "CPU" as a synonym for a "core". This is Linux's terminology. We can store a lot of important information in a dedicated per-CPU/per-core area, notably the `current` (task) variable, which is needed very often. It is clearly a global variable insofar as the kernel code running on the CPU is concerned. We thus want to access it with as few memory accesses as possible. In our current solution with segmentation, we are reading the variable with just a single instruction. This was made possible because the `gs` register directly stores a pointer to the beginning of the dedicated storage region, and the offset of the `task_struct` from that region is known. An astute reader can clearly make out that this mechanism is more efficient than the earlier method that used a redirection via the `thread_info` structure. The slower redirection-based mechanism is still used in architectures that do not have support for segmentation.

There are many things to be learned here. The first is that for something as important as the current task, which is accessed very frequently, and is often on the critical path, there is a need to devise a very efficient mechanism. Furthermore, we also need to note the diligence of the kernel developers in this regard and appreciate how much they have worked to make each and every mechanism as efficient as possible – save memory accesses wherever and whenever possible. In this case, several conventional solutions are clearly not feasible such as storing the `current task` pointer in CPU registers, a privileged/model-specific register (not a portable choice), or even a known memory address. The issue with storing this pointer at a known memory address is that it significantly limits our

---

[2]In an inline function, the code of the function is expanded at the point of invocation. There is no function call and return. This method enhances the performance of very small functions.

flexibility in using the virtual address space. This may create portability issues across architectures. As a result, the developers chose the segmentation-based method for x86 hardware.

There is a small technicality here. We need to note that different CPUs (cores on a machine) will have different per-CPU regions. This, in practice, can be realized very easily with this scheme because different CPUs have different segment registers. We also need to ensure that these per-CPU regions are aligned to cache line boundaries. This means that a cache line is uniquely allocated to a per-CPU region – there are no overlaps. If this is the case, we will have a lot of *false sharing* misses across the CPUs, which will prove to be detrimental to the overall performance. Recall that *false sharing* misses are an artifact of cache coherence. A cache line may end up continually bouncing between cores if they are interested in accessing different non-overlapping chunks of that same cache line.

3.1.6   Task Priorities

Now that we have discussed the basics of the kernel stack, task states and basic bookkeeping data structures, let us move on to understanding how we specify the priorities of tasks. This is an important input to the scheduler.

Task types	Range
Real time priorities	0-99
User task priorities	100-139

Table 3.2: Linux task priorities

Linux uses 140 task priorities. The priority range as shown in Table 3.2 is from 0 to 139. The priorities 0-99 are for real-time tasks. These tasks are for mission-critical operations, where deadline misses are often not allowed. The scheduler needs to execute them as soon as possible.

The reason we have 100 different priorities for such real-time processes is because we can have real-time tasks that have different degrees of importance. We can have some that have relatively "soft" requirements, in the sense that it is fine if they are occasionally delayed. Whereas, we may have some tasks where no delay is tolerable. The way we interpret the priority range 0-99 is as follows. In this space, 0 corresponds to the least priority real-time task and the task with priority 99 has the highest priority in the overall system.

Some kernel threads run with real-time priorities, especially if they are involved in important bookkeeping activities or interact with sensitive hardware devices. Their priorities are typically in the range of 40 to 60. In general, it is not advisable to have a lot of real-time tasks with very high priorities (more than 60) because the system tends to become quite unstable. The reason is that the CPU time is completely monopolized by these real-time tasks, resulting in the rest of the tasks, including many OS tasks, not getting enough time to execute. Hence, a lot of important kernel activities get delayed.

Now for regular user-level tasks, we interpret their priority slightly differently. In this case, higher the priority number, lower is the actual priority. This basically means that in the entire system, the task with priority 139 has the

least priority. On the other hand, the task with priority 100 has the highest priority among all regular user-level tasks. It still does not have a real-time priority but among non-real-time tasks it has the highest priority. The important point to understand is that the way that we understand these numbers is quite different for real-time and non-real-time tasks. We interpret them in diametrically opposite manners in both the cases (refer to Figure 3.4).
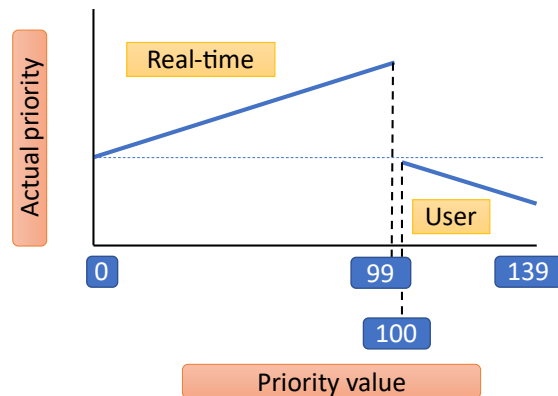
Figure 3.4: Real time priority vs the priority number (value)

3.1.7 Computing Actual Task Priorities

Listing 3.3: The `thread_info` structure.
source : kernel/sched/core.c#L2106

```
else if (rt_policy(policy))
    prio = MAX_RT_PRIO - 1 - rt_prio;
else
    prio = NICE_TO_PRIO(nice);
```

There are two concepts here. The first is the number that we assign in the range 0-139, and the second is the way that we interpret the number as a task priority. It is clear from the preceding discussion that the number is interpreted differently for regular and real-time tasks. However, if we consider the kernel, it needs to resolve the ambiguity and use a single number to represent the priority of a task. We would ideally like to have some degree of monotonicity. Ideally, we want that either a lower value should always correspond to a higher priority or the reverse, but we never want a combination of the two in the actual kernel code. This is exactly what is being rectified in the code snippet shown in Listing 3.3. We need to note that there are historical reasons for interpreting user and real-time priority numbers at the application level differently, but in the kernel code this ambiguity needs to be resolved and monotonicity needs to be ensured.

In line with this philosophy, let us consider the first `else if` condition that corresponds to real-time tasks. In this case, the value of `MAX_RT_PRIO` is 100. Hence, the range [0-99] gets translated to [99-0]. This basically means that lower the value of `prio`, greater the priority. We would want user-level priorities

to be interpreted similarly. Hence, let us proceed to the body of the `else` statement. Here, the macro `NICE_TO_PRIO` is used. Before expanding the micro, it is important to understand the notion of being *nice* in Linux.

The default user-level priority associated with a regular task is 120. Given a choice, every user would like to raise the priority of her task to be as high as possible. After all everybody wants their task to finish quickly. Hence, the designers of Linux decided (rightfully so) to not give users the ability to arbitrarily raise the priorities of their tasks. Instead, they allowed users to do the reverse, which was to reduce the priority of their tasks. It is a way to be nice to others. There are many instances where it is advisable to do so. For instance, there are many tasks that do routine bookkeeping activities. They are not very critical to the operation of the entire system. In this case, it is a good idea for users to be courteous and let the operating system know that their task is not very important. The scheduler can thus give more priority to other tasks. There is a formal method of doing this, which is known as the *nice* mechanism. As the name suggests, the user can increase the priority value from 120 to any number in the range 121-139 by specifying a *nice* value. The *nice* value in this case is a positive number, which is added to the number 120. The final value represents the priority of the process. The macro `NICE_TO_PRIO` implements this addition – it adds the *nice* value to 120.

There is a mechanism to also have a negative nice value. This mechanism is limited to the superuser, who is also known as the root user in Linux-based systems. This user has some additional privileges such as being able to access all the files and being able to raise the priority of processes. However, she does not have kernel-level privileges. She is supposed to play the role of a system administrator, and can specify a negative nice value that is between -1 and -20. Note that this mechanism cannot be used to raise the priority of a regular user-level process to that of a real-time process. We are underscoring the fact that regular users who are not superusers cannot access this facility. Their nice values are strictly positive and are in the range [1-19].

Now we can fully make sense of the code shown in Listing 3.3. We have converted the user or real-time priority to a single number `prio`. Lower it is, greater is the actual priority. This number is henceforth used throughout the kernel code to represent actual task priorities. We will observe that when we discuss schedulers, the process priorities will be very important and shall play a vital role in making scheduling decisions.

3.1.8 `sched_info`

Listing 3.4: The `sched_info` structure.
source : include/linux/sched.h#L377

```c
/* # of times we have run on this CPU: */
unsigned long pcount;

/* Time spent waiting on a runqueue: */
unsigned long long   run_delay;

/* Timestamps: */

/* When did we last run on a CPU? */
```

```
unsigned long long last_arrival;

/* When were we last queued to run? */
unsigned long long  last_queued;
```

The class `sched_info` (shown in Listing 3.4) contains some meta-information about the overall scheduling process. The variable `pcount` denotes the number of times this task has run on the CPU. `run_delay` is the time spent waiting in the runqueue. The *runqueue* is a structure that stores all the tasks whose status is TASK_RUNNING.[3] As we have discussed earlier, this includes tasks that are currently running on CPUs as well as tasks that are ready to run. Then we have a bunch of timestamps. The most important timestamps are `last_arrival` and `last_queued`, which store when a task last ran on the CPU, and it was last queued to run, respectively. In general, the unit of time within a CPU is either in milliseconds or in jiffies (refer to Section 2.1.4).

3.1.9 Memory Management
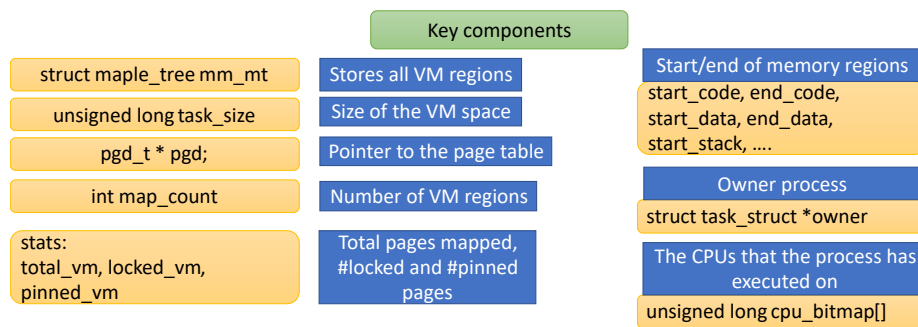
Figure 3.5: The key components of a process's address space (these fields belong to the `mm_struct` field of the `task_struct`)

A process typically owns a lot of memory space. It owns numerous virtual memory regions and a page table. Each page table entry stores the virtual-to-physical mapping and a lot of additional information. We have already looked at the protection bits stored in each entry. In addition, performance and correctness-related hints are also stored, such as information related to page locking and pinning (see [Corbet, 2014]).

For instance, we may want to *lock* a set of pages in memory and not swap them to the disk unless there is a system emergency. This will eliminate page faults for those set of pages at the cost of disadvantaging accesses to other pages. On the other hand, we can also *pin* the pages in memory, which introduces a different kind of restriction. It does not allow the kernel to move the pages around in memory, i.e., change the virtual to physical mapping. It directs the kernel to keep the page at a single physical location and not relocate it over time as is the case with regular pages. For example, we would want the pages

---

[3]We use the same terminology as the Linux kernel and omit the space between the words "run" and "queue".

of the page table to be *pinned*. This way, we would exactly know where they are, and this information will remain the same throughout the execution of the process. The kernel can access the page table using its physical address. There is no need to issue a lookup operation to find where the page table of a given process is currently located. This approach also reduces TLB misses because a lot of the mappings do not change.

The kernel uses a very elaborate data structure known as `struct mm_struct` to maintain all memory-related information of this nature as shown in Figure 3.5. The core data structure, `mm_struct`, has many fields as shown in the figure.

As we just discussed, one of the key roles of this structure is to keep track of the memory map (refer to Section 2.2.1). This means that we need to keep track of all the virtual memory regions that are owned by a process. The kernel uses a dedicated structure known as a maple tree that keeps track of all these regions. It is a sophisticated variant of a traditional B+ tree (see Appendix C). Each key in the maple tree is actually a 2-tuple: starting and ending address of the region. The key thus represents a range. In this case, the keys (and their corresponding ranges) are non-overlapping. Hence, it is easily possible to find which region a virtual address is a part of by just traversing the maple tree. This takes logarithmic time.

Along with the memory map, the other important piece of information that the `mm_struct` stores is a pointer to the page table (`pgd`). Linux uses a multi-level page table, where each entry contains a lot of information – this is necessary for address translation, security and high performance. Readers should note the high level of abstraction here. The entire page table is referenced using just a single pointer: `pgd_t* pgd`. All the operations performed on the page table require nothing more than this single pointer. This is a very elegant design pattern and is repeated throughout the kernel.

Next, the structure contains a bunch of statistics about the total number of pages, the number of locked pages, the number of pinned pages and the details of different memory regions in the memory map. For example, this structure stores the starting and ending virtual addresses of the code, data and stack sections. Next, the id of the owner process (pointer to a `task_struct`) is stored. There is a one-to-one correspondence between a process and its `mm_struct`.

The last field `cpu_bitmap` is somewhat interesting. It is a bitmap of all the CPUs on which the current task has executed in the past. For example, if there are 8 CPUs in the system, then the bitmap will have 8 bits. If bit 3 is set to 1, then it means that the task has executed on CPU #3 in the past. This is an important piece of information because we need to understand that if a task has executed on a CPU in the past, then most likely its caches will have warm data. In this case "warm data" refers to data that the current task is most likely going to use in the near future. Given that programs exhibit temporal locality, they tend to access data that they have recently accessed in the past. This is why it is a good idea to record the past history of the current task's execution. Given a choice, it should always be relocated to a CPU on which it has executed in the recent past. In that case, we are maximizing the chances of finding data in the caches, which may still prove to be useful in the near future.

3.1.10 Storing Virtual Memory Regions

Let us now address the first problem: storing a list of all the virtual memory regions owned by the process. Recall that when we had introduced the memory map of a process, we had observed that there are a few contiguous regions interspersed with massive holes. The memory map, especially in a 64-bit system, is a very sparse structure. In the middle of the large sparse areas, small chunks of virtual memory are used by the process. Hence, any data structure that is chosen needs to take this sparsity into account.

Many of the regions in the memory map have already been introduced such as the heap, stack, text, data and bss regions. In between the stack and heap there is a huge empty space. In the middle of this space, some virtual memory regions are used for mapping files and loading shared libraries. There are many other miscellaneous entities that are stored in the memory map such as handles to resources that a process owns. Hence, it is advisable to have an elaborate data structure that keeps track of all the used virtual memory regions regardless of their actual purpose. Each virtual memory region can have the same level of memory protection, read/write policies and methods to handle page faults. Instead of treating each virtual page distinctly, it is a good idea to group them into regions and assign common attributes and policies to each region. Hence, we need to design a data structure that answers the following question.

> **Question 3.1.1**
>
> Given a virtual memory address, find the virtual memory region that it is a part of.

Listing 3.5: The `vm_area_struct` structure.
source : include/linux/mm_types.h#L535

```
struct vm_area_struct {
    unsigned long vm_start, vm_end;
    struct mm_struct *vm_mm; /* Pointer to the address space
        */
    struct list_head anon_vma_chain; /*List of all anon VM
        regions */
    struct file *vmfile;
}
```

Listing 3.5 shows the code of `vm_area_struct` that represents a contiguous virtual memory region. As we can see from the code, it maintains the details of each virtual memory (VM) region including its starting and ending addresses. It also contains a pointer to the parent `mm_struct`. For understanding the rest of the fields, let us introduce the two kinds of memory regions in Linux: anonymous and file-backed.

Anonymous memory region These are many memory regions that are not mirrored or copied from a file such as the stack and heap. These memory regions are created during the execution of the process and store dynamically allocated data. Hence, these are referred to as *anonymous* memory regions. They have a dynamic existence, and are not linked to specific sections in a binary or object file.

File-backed memory region These memory regions are copies of chunks of data stored in files (sequence of bytes stored on a storage device). For example, we can have memory-mapped files, where a part of the virtual memory space is mapped to a file. This means that the contents of the file are physically copied to memory and that region is mapped to a virtual memory region. Typically, if we write to that region in memory, the changes will ultimately reflect in the backing file. This backing file is referred to as `vmfile` in `struct vm_area_struct`.

For tracking anonymous memory, there is a very elaborate data structure, which is actually a complex graph of linked lists. We will study this later when we discuss physical memory allocation in detail, especially reverse mapping. For now, it suffices to say that there is a linked list pointed to by the `anon_vma_chain` structure to store these regions. Basically, there is a pointer from `vm_area_struct` to the corresponding region in the linked list of anonymous regions.

3.1.11 The Process ID

Let us now come to one of the most important fields of `task_struct`. It is the process id or *pid*. This number uniquely identifies the task. Its type is `pid_t`, which resolves to an `unsigned int` on most architectures. Recall that every thread has a unique *pid* (process id). However, threads can be grouped, and the group has a unique thread group id (*tgid*). The *tgid* is equal to the *pid* of the leader thread. In Linux the `ps` utility lists all this information for running processes. It is equivalent to looking at all the running processes in the task manager on Microsoft Windows. Many times, we inspect the state of a process after it has finished and its *pid* has possibly been reused. For such cases, Linux provides a data structure called `struct pid` that stores all process-related information. This structure retains its information even after the process has terminated and become a zombie.

Point 3.1.3

Unfortunately, in Linux two different entities share the same name, i.e., `pid`. The number of a process has type `pid_t` and the name of the corresponding member in `task_struct` is `pid`. The structure that maintains all the information related to a process is also called `pid`. Its type is `struct pid`. This can cause a lot of confusion. We sadly do not have a choice. We need to infer the nature of the usage given the context. We adopt the following convention. When we use the term *pid*, we will be referring to `pid_t pid`. We will also use the term "pid number". Similarly, when we wish to refer to `struct pid`, we will use the term "struct pid".

Now, managing all the *pid* numbers is an important problem. Whenever a new process is started, we need to allocate a new *pid* (`pid_t`) to it. Likewise, whenever a process's state is destroyed, we need to deallocate its *pid*. The file `proc/sys/kernel/pid_max` stores the maximum number of *pid*s we can have in the system. Its default value is 32,768.

Next, we need to answer the following questions while managing *pid*s (pid numbers).

1. How do we locate the `struct pid` structure for a given *pid*?

2. How do we find the next free *pid*?

3. How do we quickly deallocate a *pid*?

4. How do we find if a *pid* is allocated or not?

Mapping a pid to a pid Structure

Let us answer the first question here. We shall defer answering the rest of the questions until we have explained some additional concepts. For mapping a *pid* to a `struct pid`, the kernel uses a radix tree (see Appendix C).

A natural question that will arise here is why not use a hash table? The kernel developers conducted a lot of experiments and tested a lot of data structures. They found that most of the time, process ids (*pid*s) share prefixes: their more significant digits. This is because, most of the time, the processes that are active have a roughly similar set of *pid*s (created at roughly the same time). As a result, if let's say we have looked up one process's entry, the relevant part of the radix tree is still present in the processor's caches. The process of looking up the `struct pid` of a related process can use some of this information to quickly realize a lookup. Hence, in practice, such radix trees were found to be faster than hash tables.

3.1.12   Namespaces

Containers

Traditional cloud computing is quickly being complemented with many new technologies: microservices, containers and serverless computing. We shall focus on them in Chapter 8. The basic idea is that a virtual machine (VM) is a full virtualized environment where every processor resource including the CPUs and memory are virtualized. These VMs can be suspended, moved to a new machine and restarted. However, this is a heavy-duty solution. Containers on the other hand are lightweight solution where the environment is not virtualized. A container is used to create a small isolated environment within a machine that is a "mini-virtual machine". Processes within a container perceive an isolated environment. They have their own set of processes, network stack and file system. They also provide strong security guarantees.

Almost all modern versions of Linux support *containers* such as Docker[4], Podman[5] and LXC[6]. A container is primarily a set of processes that own file and network resources. These are exclusive to the container that allow it to host a custom environment. For example, if the user has spent a lot of effort in creating a custom software environment, she would not like to again install the same software programs on another machine. Along with re-installing the same software, configuring the system is quite cumbersome. A lot of environment

---

[4]https://www.docker.com/
[5]https://www.docker.com/
[6]https://linuxcontainers.org/

variables need to be set and a lot of script files need to be written. Instead of repeating this same sequence of burdensome steps repeatedly, it is a better idea to create a custom file system, mount it on a Docker container and simply distribute the Docker container. All that one needs to do on a remote machine is just run the container. No additional effort is involved in installing software or configuring the runtime. This saves a lot of time. Given that containers have their virtual network interfaces, a lot of the overhead related to network configuration is also reduced.

Our focus in this chapter is the set of processes in a container. They are isolated from the rest of the system. We shall shortly see that the notion of process namespaces allows the creation of such functionality. In fact, in conjunction with software such as CRIU[7], it is possible to suspend all the processes running in the container, migrate the container (along with all its constituent processes) and restart all of them on a new machine. The entire container restarts magically on a new machine, unbeknownst to all the constituent processes.

The container creates a <u>barrier</u> between its constituent set of processes and the rest of the system. This feature allows the user to securely execute a process on a remote system. It is not necessary for the user's code and the remote system to completely trust each other. Containers ensure that the process cannot do a lot of damage to the remote system as well as the remote system cannot tamper with the process's execution beyond a point.

Details of Namespaces

Let us discuss the idea of namespaces, which underlie the key process management subsystem of containers. They need to provide a virtualized process environment where processes retain their pid numbers, inter-process communication structures and state after migration.

Specifically, the kernel groups processes into *namespaces*. Recall that the processes are arranged as a tree. Every process has a parent process, and there is one global root process. Similarly, the namespaces are also hierarchically organized as a tree. There is a root namespace. Every process is visible to its own namespace and is additionally also visible to all ancestral namespaces. No process is visible to any child namespace.

> **Point 3.1.4**
>
> Every process is visible to its own namespace and is additionally also visible to all ancestral namespaces.

In this case, a *pid* (number) is defined only within the context of a namespace. When we migrate a container, we also migrate its namespace. Then the container is restarted on a remote machine, which is tantamount to re-instating its namespace. This means that all the paused processes in the namespace are activated. Given that this needs to happen unbeknownst to the processes in the container, the processes need to maintain the same *pid*s even on the new machine.

As discussed earlier, a namespace itself can be embedded in a hierarchy of namespaces. This is done for the ease of managing processes and implementing

---

[7]https://criu.org/

containers. Every container is assigned its separate namespace. It is possible for the system administrator to provide only a certain set of resources to the parent namespace. Then the parent namespace needs to appropriately partition these resources among its child namespaces. This allows for fine-grained resource management and tracking.

Listing 3.6: The `struct pid_namespace`
source : include/linux/pid_namespace.h#L19

```c
struct pid_namespace{
    /* A radix tree to store allocated pid structures */
    struct idr idr;

    /* Cache of pid structures */
    struct kmem_cache *pid_cachep;

    /* Level of the namespace */
    int level;

    /* Pointer to the parent namespace */
    struct pid_namespace *parent;

}
```

The code of `struct pid_namespace` is shown in Listing 3.6. The most important structure that we need to consider is `idr` (IDR tree). This is an annotated Radix tree (of type `struct idr`) and is indexed by the *pid*. The reason that there is such a sophisticated data structure here is because, in principle, a namespace could contain a very large number of processes. Hence, there is a need for a very fast data structure for storing and indexing them.

We need to understand that often there is a need to store additional data associated with a process. It is stored in a dedicated structure called (`struct pid`). The `idr` tree returns the `pid` structure for a given *pid* number. We need to note that some confusion is possible here given that both are referred to using the same term "pid".

Next, we have a kernel object cache (`kmem_cache`) or pool called `pid_cachep`. It is important to understand what a *pool* is. Typically, *free* and *malloc* calls for allocating and deallocating memory in C take a lot of time. There is also need for maintaining a complex heap memory manager, which needs to find a hole of a suitable size for allocating a new data structure. It is a much better idea to have a set of pre-allocated objects of the same type in an 1D array called a *pool*. It is a generic concept and is used in a lot of software systems including the kernel. Here, allocating a new object is as simple as fetching it from the pool and deallocating it is also simple – we need to return it back to the pool. These are very fast calls and do not involve the action of the heap memory manager, which is far slower. Furthermore, it is very easy to track memory leaks. If we forget to return objects back to the pool, then in due course of time the pool will become empty. We can then throw an exception, and let the programmer know that this is an unforeseen condition and is most likely caused by a *memory leak*. The programmer must have forgotten to return objects back to the pool.

To initialize the pool, the programmer should have some idea about the

maximum number of instances of objects that may be active at any given point of time. After adding a safety margin, the programmer needs to initialize the pool and then use it accordingly. In general, it is not expected that the pool will become empty because as discussed earlier it will lead to memory leaks. However, there could be legitimate reasons for this to happen such as a wrong initial estimate. In such cases, one of the options is to automatically enlarge the pool size up till a certain limit. Note that a pool can store only one kind of objects. In almost all cases, it cannot contain two different types of objects. Sometimes exceptions to this rule are made if the objects are of the same size.

Next, we store the `level` field that indicates the level of the namespace. Recall that namespaces are stored in a hierarchical fashion. This is why, every namespace has a `parent` field.

Listing 3.7: The `struct pid`
source : include/linux/pid.h#L54

```
struct upid {
    int nr;  /* pid number */
    struct pid_namespace *ns; /* namespace pointer */
};

struct pid
{
    refcount_t count;
    unsigned int level;

    /* lists of tasks that use this pid */
    struct hlist_head tasks[PIDTYPE_MAX]; /* A task group */

    /* wait queue for pidfd notifications */
    /* Array of upids , one per level */
    struct upid numbers[];
};
```

Let us now look at the code of `struct pid` in Listing 3.7. As discussed earlier, often there is a need to store additional information regarding a process, which may be used after the *pid* has been reused, and the process has terminated. The `count` field refers to the number of resources that are using the process. Ideally, it should be 0 when the process is freed. Also, every process has a default level, which is captured by the `level` field. This is the level of its original namespace.

The linked list `tasks` stores several lists of tasks. Note that `hlist_head` points to a linked list (singly-linked). It has several members. The most important members are as follows:

- `tasks[PIDTYPE_TGID]` (list of processes in the thread group)

- `tasks[PIDTYPE_PGID]` (list of processes in the process group)

- `tasks[PIDTYPE_SID]` (list of processes in the session)

We have already looked at a thread group. A process group is a set of processes that are all started from the same shell command. For example, if we

start an instance of the Chrome browser, and it starts a set of processes, they are all a part of the same process group. If the user presses Ctrl+C on the shell then the Chrome browser process and all its child processes get terminated. A *session* consists of a set of process groups. For example, all the processes created by the login shell are a part of the same session.

> **Point 3.1.5**
>
> A process may belong to a thread group. Each thread group has a thread group id, which is the *pid* of the leader process. A collection of processes and thread groups is referred to as a process group. All of them can be sent SIGINT (Ctrl+C) and SIGTST (Ctrl+z) signals from the shell. It is possible to terminate all of them in one go. A collection of process groups form a session. For example, all the processes started by the same login shell are a part of the same session.

The last field `numbers` is very interesting. It is an array of `struct upid` data structures (defined in Listing 3.7). Each `struct upid` is a tuple of the *pid* number and a pointer to the namespace. Recall that we had said that a *pid* number makes sense in only a given namespace. In other ancestral namespaces, the same process (identified with `struct pid`) can have a different process id number (*pid* value). Given that every process needs to also be listed in all ancestral namespaces, there is a need to store such ⟨*pid*,namespace⟩ tuples – one for each namespace.

IDR Tree

Each namespace has a data structure called an IDR tree (`struct idr`). IDR stands for "ID Radix" We can think of the IDR tree as an augmented version of the classical radix tree. Its nodes are annotated with additional information, which allow it to function as an augmented tree as well. Its default operation is to work like a hash table, where the key is the *pid* number and the value is the `pid` structure. This function is very easily realized by a classical radix tree. However, the IDR tree can do much more in terms of finding the lowest unallocated *pid*. This functionality is normally provided by an augmented tree (see Appendix C). The IDR tree is a beneficial combination of a radix tree and an augmented tree. It can thus be used for mapping *pid*s to `pid` structures and for finding the lowest unallocated *pid* number in a namespace in logarithmic time.

A node in the IDR tree is an `xa_node`, which typically contains an array of 64 pointers. Each entry can either point to another internal node (`xa_node`) or an object such a `struct pid`. In the former case, we are considering internal nodes in the augmented tree. The contiguous key space assigned to each subtree is split into non-overlapping regions and assigned to each child node. The leaves are the values stored in the tree. They are the objects stored in the tree (values in the key-value pairs). We reach an object (leaf node) by traversing a path based on the digits in the key.

Let us explain the method to perform a key lookup using the IDR tree. We start from the most significant MSB bits of the *pid*, and gradually proceed towards the LSB bit. This ensures that the leaves of the tree that correspond to

unique *pid*s are in sorted order if we traverse the tree using a preorder traversal. Each leaf (`struct pid`) corresponds to a valid *pid*.

Definition 3.1.1 IDR Tree

An IDR tree is a key-value storage structure, where the key is the *pid* number and the value is its corresponding `struct pid`. The values are stored in the leaves of the tree. It is a combination of a radix tree and an augmented tree.

Using the IDR Tree as an Augmented Tree

Each `xa_node` in the IDR tree stores a bit vector (`marks`) and an array of pointers (`slots`). Typically, both have 64 entries each. If the ith bit is 1 in the bit vector, then the ith subtree has a free entry: unallocated *pid* in its assigned range. If it is 0, then it means that the ith subtree does not have any unallocated *pid*s. The advantage of such augmented trees is that the entire subtree can be skipped if it does not have any free entries.

Let us explain with an example shown in Figure 3.6. Assume there are five allocated *pid*s: 0, 1, 3, 4 and 7. Their binary representations are 000, 001, 011, 100 and 111, respectively. Given that we start from the most significant bit, we can create a radix tree as shown in Figure 3.6. The internal nodes are shown as ovals and the leaf nodes corresponding to `struct pid`s are shown as rectangles.
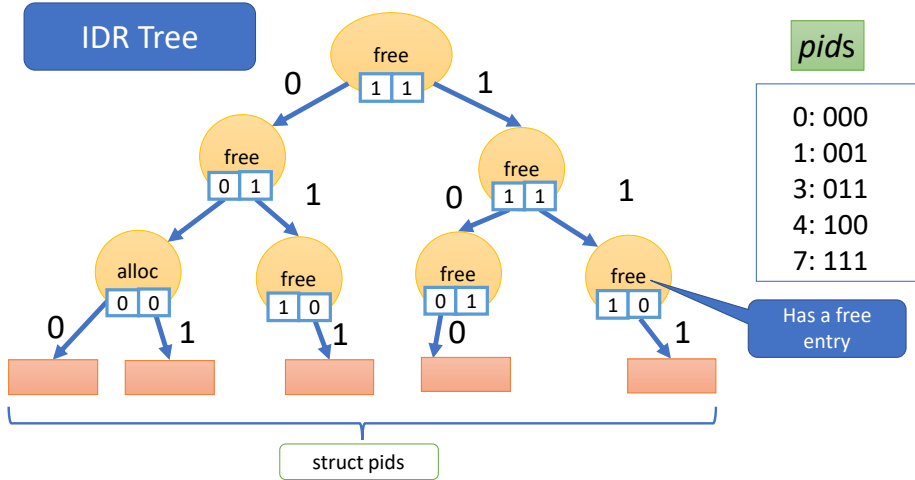
Figure 3.6: Example of an IDR tree

Trivia 3.1.1

It is important to note that we do not store a bit vector explicitly at one place. The bit vector is distributed across all the internal nodes at the second-last level. Nodes at this level point to the leaves.

Scanning every bit sequentially in the bit vector `marks` stored in an `xa_node` can take a lot of time (refer to Figure 3.7). If it is a 64-bit wide field, we need
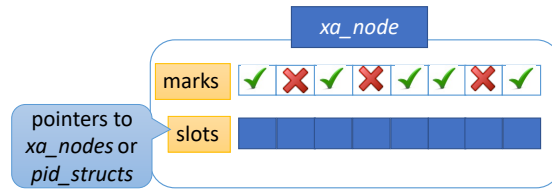
Figure 3.7: The `slots` and `marks` arrays in an `xa_node`

to run a `for` loop that has 64 iterations. Fortunately, on x86 machines, there is an instruction called `bsf` (bit scan forward) that returns the position of the first (least significant) 1. This is a very fast hardware instruction that executes in 2-3 cycles. The kernel uses this instruction to almost instantaneously find the location of the first 1 bit (free bit).

Once a free bit is found, it is set to 0, and the corresponding *pid* number is deemed to be allocated. This is equivalent to converting a 1 to a 0 in a augmented tree (see Appendix C). There is a need to traverse the path from the leaf to the root and change the status of nodes accordingly. Similarly, when a *pid* is deallocated, we convert the corresponding bit from 0 to 1, and appropriate changes are made to the nodes in the path from the leaf to the root.

Allocating a `pid` Structure

Let us now look at the process of allocating and registering a new process id. We start with invoking the `alloc_pid` function defined in kernel/pid.c. The first step is to find a free `struct pid` structure from the pool of `pid` structures. This is always the method of choice because it is very fast and a pool also helps detect memory leaks. There is no additional overhead of `malloc` calls.

The next step is to allocate a *pid* number in each namespace that the process is a part of. This includes its default namespace as well as all ancestral namespaces. The idea is that a namespace can potentially see all the processes in its descendant namespaces. However, the reverse is not possible. Hence, there is a need to visit all ancestral namespaces, access their respective IDR trees, find the smallest unallocated *pid* number, and then create a mapping between the *pid* number and the newly allocated `pid` structure. Note that a *pid* number is only defined within its namespace, not in any other namespace. Hence, there is a need to iteratively visit every ancestral namespace and make an entry in its respective IDR tree.

3.1.13 File System, I/O and Debugging Fields

A *file* is a contiguous array of bytes that is stored on a storage device such as a hard disk or a flash drive. Files can have different formats. For example, video files (.mp4) look very different from Word documents (.docx). However, for the operating system, a file is simply an array of bytes. There are different kinds of file systems. They organize their constituent files and directories differently based on considerations such as read-write patterns, sequential vs random accesses, need for reliability, and so on. At the task level, all that is needed is a generic handle to a file system and a list of open files.

Along with that, we also need to understand that the file system is typically coupled with a storage device. Linux defines most storage devices like hard disks and flash drives to be block-level storage devices – their atomic storage units are *blocks* (512 B to 4 KB). It is necessary to also maintain some information regarding the I/O requests that have been sent to different block devices. Linux also defines character devices such as the keyboard and mouse that typically send a single character (a few bytes) at a time. Whenever, some I/O operation completes or a character device sends some data, it is necessary to call a signal handler. Recall, that a signal is a message sent from the operating system to a task. The signal handler is a specialized function that is registered with the kernel.

The fields that store all this information in the `task_struct` are as follows.

Listing 3.8: I/O and signal-handling fields in `task_struct`

```
/* Pointer to the file system */
struct fs_struct *fs;

/* List of files opened by the process */
struct files_struct *files;

/* List of registered signal handlers */
struct signal_struct *signal;

/* Information about block devices. bio stands for block
    I/O*/
struct bio_list *bio_list;

/* I/O device context */
struct io_context *io_context;
```

The PTrace Mechanism

There is often a need for a parent process to observe and control the execution of a child process. This needs to be done for debugging purposes. However, there are other security-related applications also. In many cases, especially when we do not trust the child process, it is necessary to keep a tab on its activity and ensure that from a security point of view everything is alright – the child process is not doing something that it is not supposed to do. This mechanism is known as *tracing*.

In this mechanism, a process can be *traced* by another process (the tracing process). The `task_struct` has a field called `unsigned int ptrace`. The flags in this field define the kind of tracing that is allowed.

The general idea is as follows. Whenever there is an event of interest such as a system call, then the task stops and a SIGTRAP signal is sent to the tracing process. We are quite concerned about system calls because this is the primary mechanism by which a process interacts with the operating system. If a user's intent is malicious, then this will manifest via potentially erroneous or mala fide system calls. As a result, it is important to thoroughly scrutinize the interaction of a process with the kernel.

In this case, the tracing process runs the SIGTRAP signal handler. In the signal handler, it inspects the state of the traced process (it has the permission

to do so) and looks at all the system call parameters. At this stage, it is also possible to change the system call parameters. This is especially interesting when we are trying to put in additional information for the purposes of debugging. Also, sometimes we would like to send specific information to the kernel such that it can track the information flow emanating from a traced process much better. This is why, modifying system call arguments can be very useful. Furthermore, if system calls can potentially do something malicious, then it makes a lot of sense to create more innocuous forms of them such that their potential to do damage is limited.

3.2 Process Creation and Destruction

The notion of creation and destruction of threads, processes and tasks is vital to the execution of any system. There needs to be a seamless mechanism to create and destroy processes. The approach that Linux takes may seem unconventional, but it is a standard approach in all Unix-like operating systems. There are historical reasons and over time programmers have learned how to leverage it to design efficient systems. Other operating systems like Windows use other mechanisms. This model is simple and is also intuitive, once understood properly.

The kernel defines a few special processes notably the *idle process* that does nothing and the *init* process. At the outset, the kernel starts a single process to boot the operating system. Once booting is done, this booting process becomes the idle process (also known as the *swapper*). Its *pid* is 0. It spawns the *init* process that starts as a kernel process. *init* transitions to the user mode and spends the rest of its life as a user process. Its *pid* is 1. It acts as the mother process of all user space processes. Its role is to spawn all user-level processes. Recall that processes are arranged as a tree. In this case, *init* is the root of the tree that comprises all user space processes. Its parent is the idle process. The idle process spawns another process called *kthreadd* ($pid = 2$), which acts like the mother process for all kernel threads. A kernel thread exclusively does work for the kernel. It never transitions to user mode. *kthreadd* is the root of the tree that comprises all kernel processes.

3.2.1 The Fork Mechanism

The `fork` system call is arguably the most famous in this space. It creates a `clone` of a running process at the point that it is called. Its role is to create a straightforward copy of the running process, which involves copying its entire memory and runtime state. The process that executed the `fork` call is henceforth the parent process, and the process that was created as a result of the call becomes its child. The child inherits a copy of the parent's complete memory and execution state. A twin of the parent is created. We need to note that after returning from the `fork` call, the parent and the newly created child are separate entities. They do not share any resources, and the two processes are free to go their separate ways. For all practical purposes, they are separate processes and are free to choose their execution paths. Right after the `fork` call, their memory spaces just happen to be exact copies of each other and their program counters have the same values. Note that they have separate virtual

and physical address spaces, and never share frames that any processes writes to (after the `fork` call).

Before we delve more into the details of the `fork` system call, let us revisit the *init* process. The systems boots by calling the function `start_kernel` (defined in init/main.c). Its job is to initialize the kernel as well as all the connected devices. Think of it as the kernel's *main* function. After doing its job, it forks the *init* process. The *init* process thus begins its life inside the kernel. However, it quickly transitions to the user mode and remains a user-mode process subsequently. This is achieved using another kernel mechanism (system call) known as `execve`, which we shall discuss later. This is a rare instance of a process being born in the kernel and living its life as a regular user-mode process. Subsequently, every user-mode process is born (created) by either forking the *init* process or another user process. Note that we are pretty much creating a tree of processes that is rooted at the *init* process.

Once the *init* process has been created, and all the necessary user-level processes and *kthreadd* have been created, the boot process ends. We can then interact with the system and launch applications. Note that all processes are created using exactly the same mechanism. Some process that has already been created is forked to create an application process. After creation, a child process is not bound to use the state that it copied from its parent or execute the same code. It is an independent process and it is free to execute any piece of code, as long as it has the requisite permissions.

Listing 3.9: Example of the `fork` system call

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4
5   int main( void ) {
6       int pid = fork();
7
8       if (pid == 0) {
9           printf( "I am the child \n" );
10      } else {
11          printf( "I am the parent: child = %d\n", pid );
12      }
13  }
```

An example of using the `fork` system call is shown in Listing 3.9. Here, the `fork` library recall is used that encapsulates the `fork` system call. The `fork` library call returns a process id (variable `pid` in the code) after creating the child process.

It is clear that inside the code of the forking procedure, a new process is created, which is a child of the parent process that made the `fork` call. It is a perfect copy of the parent process. It inherits the parent's code as well as its memory state. In this case, *inheriting* means that all the memory regions and the state are fully copied and the copy is assigned to the child. For example, if a variable `x` is defined to be 7 in the code before executing the `fork` call, then after the call is over and the child is created, both of the processes can read `x`. They will see its value to be 7. However, there is a point to note here. The variable `x` is different for both the processes even though it has the same value,

i.e., 7. This means that if the parent changes x to 19, the child will still read it to be 7 because it has its own private copy of x. We need to appreciate that the child gets a copy of the value of x, not a reference to the parent's x variable. Even though the name x is the same across the two processes, the variables themselves are different.

Now that we have clarified the meaning of copying the entire memory space, let us look at the return value. Both the child and the parent will return from the fork call. The weird part of the whole story is that the child process will appear to return from the fork call even though it did not invoke it. It clearly did not exist when the fork call was invoked. Regardless of this small non-intuitive anomaly, the child will "appear" to return from the fork call. This is the fun and tricky part. When the child is created deep in the kernel's process-cloning logic, a complete task_struct along with all of its accompanying data structures is created. The memory space is fully copied including the register state and the value of the return address. The state of the task is also fully copied. Since all the addresses are virtual, creating a copy does not hamper correctness. Insofar as the child process is concerned, all the addresses that it needs are a part of its address space. It is, at this point of time, indistinguishable from the parent. The same way that the parent will eventually return from the fork call, the child also will. The child will get the return address from either the register or the stack (depending upon the architecture). This return address, which is virtual, will be in its own address space. Given that the code is fully copied, the child will place the return value in its private variable pid and start executing Line 8 in Listing 3.9. Also refer to Figure 3.8.
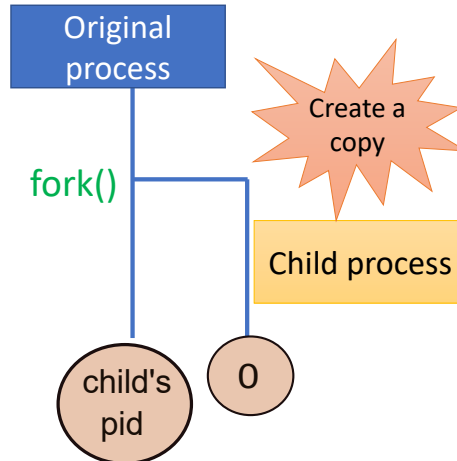
Figure 3.8: Forking a child process

Herein lies the brilliance of this mechanism – the parent and child are returned different values.

Point 3.2.1

The child is returned 0 and the parent is returned the *pid* of the child.

This part is crucial because it helps the rest of the code differentiate between

the parent and the child. A process knows whether it is the parent process or the child process from the return value: 0 for the child and the child's *pid* for the parent. Subsequently, the child and parent go their separate ways. Based on the return value of the `fork` call, the `if` statement is used to differentiate between the child and parent. Both can execute arbitrary code beyond this point and their behavior can completely diverge. In fact, we shall see that the child can completely replace its memory map and execute some other binary. However, before we go that far, let us look at how the address space of one process is completely copied. This is known as the copy-on-write mechanism.

Copy-on-Write

(a) Parent and child sharing a page

(b) Separate mappings after a copy-on-write
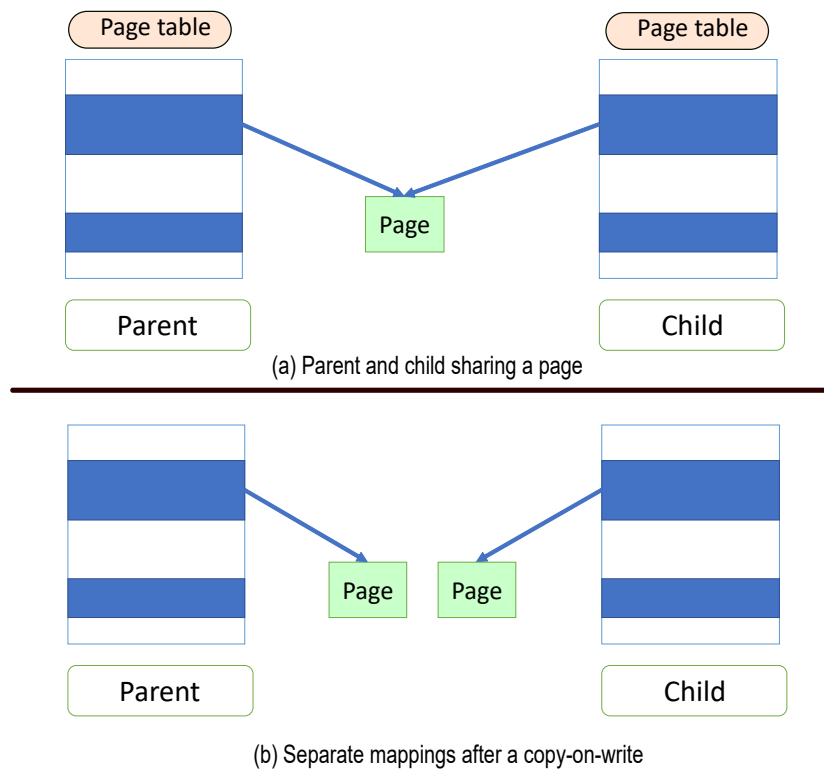
Figure 3.9: The copy-on-write mechanism

Figure 3.9(a) shows the copy-on-write (CoW) mechanism. To begin with, we just copy the page tables. The child inherits a complete copy of the parent's page table even though it has a different address space. This mechanism ensures that the same virtual address in both the child and parent's virtual address spaces points to the same physical address. No memory is wasted in the copying process and the size of the memory footprint remains exactly the same. This is a fast mechanism. Copying the page table implies copying the entire memory space including the text, data, bss, stack and heap sections. Other than the return value of the `fork` call, nothing else differentiates the child and parent. Note that this is an implementation hack that just makes the `fork` process fast. As

long as there are no write operations after the `fork` call, this mechanism will work. Since we are only performing read operations, we are only interested in getting the correct values – the same will be obtained. However, the moment there is a write operation, initiated by either the parent or the child after the `fork` operation, some additional work needs to be done.

Let us understand our constraints. We do not share any variables between the parent and the child. As we have discussed earlier, if a variable `x` is defined before the `fork` call, after the call it actually becomes two variables: `x` in the parent's address space and `x` in the child's address space. This cannot be achieved by just copying the page table of the parent. We clearly need to do more if there is a write.

This part is shown in Figure 3.9(b). Whenever there is a write operation that is initiated by the parent or the child, we create a new copy of the data for the writing process. This is done at the page level. This means that a new physical copy of the frame is created and mapped to the respective virtual address space. This requires changes in the TLB and page table of the writing process. The child and parent now have different mappings in their TLBs and page tables. The virtual addresses that were written to now point to different physical addresses. Assume that the child initiated the write, then it gets a new copy of the frame and appropriate changes are made to its TLB and page table to reflect the new mapping. Subsequently, the write operation is realized. To summarize, the child writes to its "private" copy of the page. This write is not visible to the parent.

As the name suggests, this is a copy-on-write mechanism where the child and parent continue to use the same physical page (frame) until there is a write operation initiated by either one. This approach can easily be realized by just copying the page table, which is a very fast operation. The moment there is a write, there is a need to create a new copy of the corresponding frame, assign it to the writing process, and then proceed with the write operation. This increases the performance overheads when it comes to the first write operation after a `fork` call; however, a lot of this overhead gets amortized and is seldom visible.

There are several reasons for this. The first is that the parent and child may not subsequently write to a large part of the memory space such as the code and data sections. In this case, the copy-on-write mechanism will never get activated. The child may end up overwriting its memory image with that of another binary and this will end up erasing its entire memory map. There will thus be no need to invoke the CoW mechanism. Furthermore, lazily creating copies of frames as and when there is a demand, distributes the overheads over a long period of time. Most applications can absorb this overhead very easily. Hence, the `fork` mechanism has withstood the test of time.

Tracking Page Accesses

A question that naturally arises here is how do we know if a page has been written to? We need to cleverly use the permission bits in the TLB and page table. Recall that every TLB or page table entry has a few permission bits that specify whether the page can be written to or not. In this case, we mark all the pages as read-only (after a `fork` operation). Whenever there is a write access, a fault will be generated, which the kernel can detect. The kernel will quickly

detect that it is a fake page fault that was deliberately induced to track page accesses. It is a page protection fault, which arose because pages' protection bits were set to read-only. There is a need to perform a copy-on-write and reset the read-only status for both the parent's and child's versions of the page.

Let us now delve into the fine print. It is possible that the parent process already has some pages such as code pages, which are meant to be always read-only. Their `READONLY` bit would have been set even before the `fork` call. This information needs to be preserved, otherwise we may erroneously reset the read-only status of such pages on a copy-on-write. Hence, modern systems have another bit, which we shall refer to as `P2`. Whenever a process is forked, we set the value of `P2` to 1 for all the pages that belong to either the parent or the child. This bit is set in the page tables.

Whenever, a process tries to write to a page whose `READONLY` bit is set to 0 (can write in normal circumstances) and `P2` is set to 1, we realize that we are trying to write to a page that has been "copied". This page was normally meant to be written because its `READONLY` bit is not set. However, its `P2` bit was set because we wish to trap all write accesses to this page. Hence, the copy-on-write mechanism needs to be invoked and a new copy of the page needs to be created. Subsequently, we can set the `P2` bits of the corresponding pages in both the parent's and child's page tables to 0. The need to track write accesses for this page is not there anymore. A separate copy has already been created and the parent and child can happily perform read and write operations on their respective private copies of the page.

Details

We would like to draw the reader's attention to the file in the kernel that lists all the supported system calls: include/linux/syscalls.h. It has a long list of system calls. However, the system calls of our interest are `clone` and `vfork`. The `clone` system call is the preferred mechanism to create a new process or thread in a thread group. It is extremely flexible and takes a wide variety of arguments. However, the `vfork` call is optimized for the case when the child process immediately makes an `exec` call to replace its memory image. In this case, there is no need to fully initialize the child and copy the page tables of the parent. Finally, note that in a multithreaded process (thread group), only the calling thread is forked.

Inside the kernel, all of these functions ultimately end up calling the `copy_process` function in kernel/fork.c. While forking a process the `vfork` call is preferred, whereas while creating a new thread, the `clone` call is preferred. The latter allows the caller to accurately indicate which memory regions need to be shared with the child and which memory regions need to be kept private. The signature of the `copy_process` function is as follows:

```
struct task_struct* copy_process (struct pid* pid, ...  )
```

Here, the ellipses . . . indicate that there are more arguments, which we are not specifying for the sake of readability. The main tasks that are involved in copying a process are as follows:

1. Duplicate the current `task_struct`.

(a) Create new task and its accompanying `task_struct`.

(b) Set up its kernel stack.

(c) Duplicate the complete architectural state, which includes pushing the state of all the registers (general purpose, privileged and flags) to the kernel stack.

(d) Add all the other bookkeeping information to the newly created `task_struct`.

(e) Set the time that the new task has run to zero.

(f) Assign this task to a CPU, which means that when the task is fully initialized, it can run on the CPU that it was assigned to.

(g) Allocate a new *pid* for the child task in its namespace, and also its ancestral namespaces.

2. Copy all the information about open files, network connections, I/O, and other resources from the parent task.

(a) Copy all the connections to open files. This means that from now on the parent and child can access the same open file (unless it is exclusively locked by the parent).

(b) Copy a reference to the current file system.

(c) Copy all information regarding signal handlers to the child.

(d) Copy the page table and other memory-related information (the complete `struct mm_struct`).

(e) Recreate all namespace memberships and copy all the I/O permissions. By default, the child has the same level of permissions as the parent.

3. Create external relationships:

(a) Add the new child task to the list of children of the parent task.

(b) Fix the parent and sibling list of the newly added child task.

(c) Add thread group, process group and session information to the child task's `struct pid`.

3.2.2 The exec Family of System Calls

It is important to note that after returning from a `fork` operation, the child and parent process are independent entities – they can go their separate ways. For example, the child may decide to completely reset its execution state and start executing a new binary, afresh and anew. This is typically the case with many user-level processes. When we issue a command on the command line, the shell process forks and creates a child process. The *shell* is basically the program that we interact with in a terminal. It accepts user inputs and starts executing a binary specified in the command. In this case, the forked shell process decides to run the binary and replaces its memory map with the memory map of the binary that needs to be executed. This is like starting a new execution afresh.

Listing 3.10: Example of the `execv` system call

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define PWDPATH "/usr/bin/pwd"

int main( void ) {
    char *argv[2] = {"pwd",NULL};
    int pid = fork();

    if (pid == 0) {
        execv (PWDPATH,argv);
    } else {
        printf( "I am the parent: child = %d\n", pid );
    }
}
```

The `exec` family of system calls are used to achieve this. In Listing 3.10, an example is shown where the child process runs the `execv` library call. Its arguments are a null-terminated string representing the path of the executable and an array of arguments. The first argument is by default the file name – `pwd` in this case. The next few arguments should be the command-line arguments to the executable and the last argument needs to be NULL. Since we do not have any arguments, our second argument is NULL. There are many library calls in the `exec` family. All of them wrap the `exec` system call.

There are many steps involved in this process. The first action is to clean up the memory space (memory map) of a process and reinitialize all the data structures. We need to then load the starting state of the new binary in the process's memory map. This includes the contents of the text and data sections. Then there is a need to initialize the stack and heap sections, and set the starting value of the stack pointer. In general, file and network connections are preserved in an `exec` call. Hence, there is no need to modify, cleanup or reinitialize them. After the `exec` call returns, we can start executing the process from the start of its new text section. We are basically starting the execution of a new program. The fact that we started from a forked process is conveniently forgotten. This is the Linux way.

3.2.3 Kernel Threads

Linux distinguishes between user threads, I/O threads, and kernel threads.

The term "user thread" has two connotations. The first connotation is that it is a regular process that starts its life in user mode. Whenever it executes a system call or its context is switched because of an interrupt, it transitions to kernel mode. Note that no additional kernel thread is spawned. The same user thread is reused to do kernel work. Its state in its `task_struct` is changed to indicate that it is a kernel thread. There is a need to ensure that whatever data it reads or writes in kernel mode is not accessible when the thread transitions back to user mode. Two steps are taken to ensure this. The first is that in kernel mode, it is assigned a separate kernel stack, which is stored in the kernel's virtual address space. The user thread in its kernel avatar can access

data in its user-mode virtual address space and can also access data in the kernel's virtual address space. The next step is that the kernel's virtual address is kept separate. For example, on a 32-bit system the lower 3 GB of the virtual address space is reserved for user programs and the upper 1 GB represents the kernel's virtual address space. All kernel-level data structures including the kernel stacks are stored in this upper 1 GB. Unless a kernel thread is explicitly accessing user space, it accesses data structures only in kernel space. No user thread can access kernel virtual memory. It will be immediately stopped by the TLB. The TLB will quickly realize that a user process wishes to access a kernel page. This information is stored in each TLB entry. Hence, processes can keep transitioning from user mode to kernel mode, and vice versa, repeatedly, without revealing kernel data to the user, unless the information is the return value of a system call.

The other type of "user threads" are not real threads. They are purely user-level entities that are created, managed and destroyed in user space. This means that a single process (recognized by the kernel) can create and manage multiple *user* threads. This could also be a multithreaded process. Regardless of its implementation, we need to note that a single group of threads manage user threads that could be far more numerous. Consider the case of a single-threaded process P that creates multiple user threads. It partitions its virtual address space and assigns dedicated memory regions to each created *user thread*. Each such user thread is given its own stack. Process P also creates a heap that is shared between all user threads. We need to understand that the kernel still perceives a single process P. If P is suspended, then all the user threads are also suspended. This mechanism is clearly not as flexible as native threads that are recognized by the kernel. It is hard to pause processes, collect their context and restore the same context later. However, user-threading libraries have become mature. It is possible to simulate much of kernel's activities such as timer interrupts and context collection using signal handlers, kernel-level timers and bespoke assembly routines. We shall use the term pure-user threads to refer to this type of threads, which are not recognized by the kernel.

Let us now look at I/O threads and kernel threads. We need to understand that Linux has a single `task_struct` and all threads are just processes. We do not have different `task_structs` for different kinds of threads. A `task_struct` however has different fields that determine its behavior. Every task has a priority and it can be a specialized task that only does kernel work. Let us look at such variations.

I/O threads are reasonably low-priority threads that are dedicated to I/O tasks. They can be in the kernel space or run exclusively in user space. Kernel threads run with kernel permissions and are often very high-priority threads. The `PF_KTHREAD` bit is set in `task_struct.flags` if a task is a kernel thread. Kernel threads exclusively do kernel work and do not transition to user mode. Linux defines analogous functions such as `kthread_create` and `kernel_clone` to create and clone kernel threads, respectively. They are primarily used for implementing all kinds of bookkeeping tasks, timers, interrupt handlers and device drivers.

3.3 Context Switching

Let us now delve into the internals of the context switch process. The process of switching the context, i.e., suspending a running process, handling the interrupt or event that caused the process to be suspended, invoking the scheduler and loading the context of the process that the scheduler chose is a very involved process. We can end up resuming the process that was paused, or we may end up waking up another process. In either case, the core algorithm is the same.

3.3.1 Hardware Context

We need to start with understanding that every process has its hardware context. This basically means that it has a certain state in hardware, which is contained in the registers, the program counter, ALU flags, etc. All of these need to be correctly saved such that the same process can be restarted later without the process even knowing that it was swapped out. This means that we need to have a very accurate mechanism to save and restore all this information. No errors are tolerable. In the context of x86-64, let us understand the term *hardware context* in some more detail. It specifically contains the contents of the following hardware entities:

1. All the general-purpose registers including the stack pointer

2. Program counter (instruction pointer in x86)

3. Segment registers

4. Privileged registers such as CR3 (starting address of the page table)

5. ALU and floating-point unit flags

There are many other minor components of the hardware context in a large and complex processor like an x86-64 machine. We have listed the main components for the sake of readability. The key point that we need to note is that this *context* needs to be correctly stored and subsequently restored.

Let us focus on the TLB now and understand the role it plays in the context switch process. It stores the most frequently (or recently) used virtual-to-physical mappings. There is a need to flush the TLB when the process changes, because the new process will have a new virtual memory map. We do not want it to use the mappings of the previous process. They will be incorrect and this will also be a serious security hazard because now the new process can access the memory space of the older process. Hence, once a process is swapped out, at least no other user-level process should have access to its TLB contents. An easy solution is to flush the TLB upon a context switch. However, as we shall see later, there is a more optimized solution, which allows us to append the *pid* number to each TLB entry. This does not require the system to flush the TLB upon a context switch, which is a very expensive solution in terms of performance. Every process should use its own mappings. Because of the *pid* information that is present, a process cannot access the mappings of any other process. This mechanism (enforced by hardware) reduces the number of TLB misses. As a result, there is a net performance improvement.

Software Context

The page table, open file and network connections and the details of similar resources that a process uses are a part of its *software context*. This information is maintained in the process's `task_struct`. There is no need to store and restore this information upon a context switch – it can always be retrieved from the `task_struct`.

The structure of the page table is quite interesting if we consider the space of both user and kernel threads. The virtual address space of any process is typically split between user space addresses and kernel addresses. On x86 machines, the kernel addresses are located at the higher part of the virtual address range. The user space addresses are at the lower end of the virtual address range. Second, note that all the kernel threads share their virtual address space. This means that across processes, the mappings of kernel virtual addresses are identical. This situation is depicted in Figure 3.10. It helps to underscore the fact that the virtual address spaces of all user processes are different. This means that across user processes, the same virtual address maps to different physical addresses unless they correspond to a shared memory channel. However, this is not the case for the kernel region. Here, the same virtual address maps to the same physical address regardless of the user process. For kernel threads that exclusively run in kernel mode, they do not use any user space virtual address. They only use kernel space virtual addresses at the upper end of the virtual address space. They also follow the same rule – all kernel space mappings are identical across all processes.
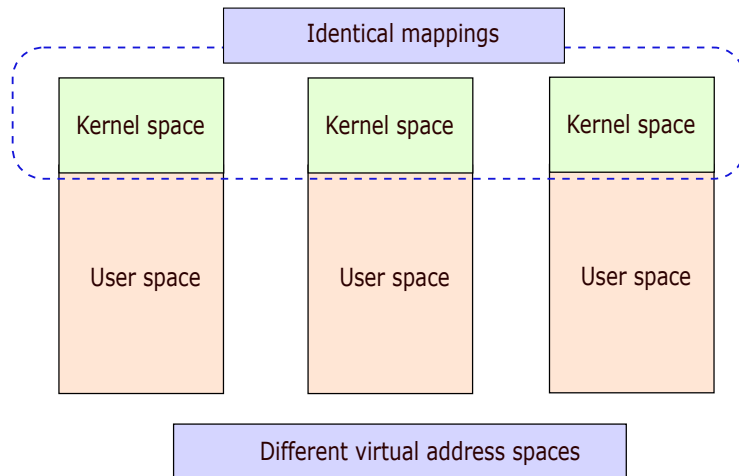
Figure 3.10: User and kernel space virtual addresses

Figure 3.10 shows that a large part of the virtual address spaces of processes have identical mappings. Hence, a part of the page table will also be common across all processes due to such identical mappings. This "kernel portion" of the page table will not change even if there is a transition from one process to another, even though the page tables themselves may change. The mappings that stand to change on a context switch are in the portion corresponding to user space addresses.

> **Point 3.3.1**
>
> The page table needs to be changed only when the user space virtual address mappings change. If there is a switch between kernel threads, there is no need to change the page table because the kernel virtual address space is the same for all kernel threads. There is no need to change the page table even if there is a transition from user mode to kernel mode. The kernel space mappings will remain the same.

> **Way Point 3.3.1**
>
> - The virtual address space of any process is split between user and kernel addresses.
>
> - All kernel threads share the virtual address space.
>
> - The kernel portion of the virtual address space has identical mappings across all processes. A kernel virtual address always maps to the same physical address. This is quite unlike user space virtual addresses, where the same virtual address typically maps to different physical addresses across processes.
>
> - Because the kernel space mappings are identical, there is often no need to change page tables on context switches (refer to Point 3.3.1).

3.3.2   Types of Context Switches

There are three types of context switches.

1. Process context switch

2. Thread context switch

3. Interrupt context switch

Process Context Switch

This is a regular context switch between processes: user or kernel. Specifically, four subtypes can be defined: kernel → kernel, kernel → user, user → kernel and user → user. As we have discussed earlier, a context switch can be triggered by three *events of interest* namely an interrupt, an exception or a system call. Additionally, we have a method of generating dummy interrupts using a timer chip because the kernel needs to execute periodically. If genuine interrupts, exceptions and system calls are not being made, there is a need to generate fake interrupts such that at least the kernel gets a chance to periodically run and do its job. After handling the event of interest, the kernel runs the scheduler. Its role is to decide whether the currently executing task has run for a long time or not, and if there is a need to suspend it and give the CPU to another task.

Whenever such an event of interest arrives, the hardware takes over and does a minimal amount of context saving. Then based on the nature of the event, it

calls the appropriate handler. If we consider the case of a timer interrupt, then the reason for the kernel's invocation is not very serious – it is a routine matter. In this case, there is no need to create an additional kernel thread that is tasked to continue the process of saving the context of the user-level thread that was executing. As discussed earlier, we can reuse the same user-level thread that was interrupted. Specifically, the same `task_struct` can be used, and the user thread can simply be run in "kernel mode". Think of this as a new avatar of the same thread, which has now ascended from the user plane to the kernel plane. This saves a lot of resources as well as time; there is no need to initialize any new data structure or create/resume any thread here.

The job of this newly converted kernel thread is to continue the process of storing the hardware context. This means that there is a need to collect the values of all the registers and store them somewhere. In general, in most architectures, the kernel stack is used to store this information. We can pretty much treat this as a *soft switch*. This is because the same thread is being reused. Its status just gets changed – it temporarily becomes a kernel thread and starts executing kernel code (not a part of the original binary though). Also, it now uses its kernel stack. Recall that the user-level stack cannot be used in kernel mode. This method is clearly performance-enhancing and is very lightweight in character. Let us now answer two key questions.

Is there a need to flush the TLB?

There is only a need to flush the TLB when the mappings change. This will only happen if the user-mode virtual address space changes (see Point 3.3.1). There is no need to flush the TLB if there is a user→kernel or kernel→kernel transition – the kernel part of the address space remains the same. Now, if we append the *pid* to each TLB entry, there is no need to remove TLB entries if the user space process changes.

Is there a need to change the page table?

The answer to this question is the same as the previous one. Whenever we are transitioning from user mode to kernel mode, there is no need to change the page table. The kernel space mappings are all that are needed in kernel mode, and they are identical for all kernel threads. Similarly, while switching between kernel threads, there is also no need. A need arises to switch the page table when we are transitioning from kernel mode to user mode, and that too not all the time. If we are switching back to the same user process that was interrupted, then also there is no need because the same page table will be used once again. We did not switch it while entering kernel mode. A need to switch the page table arises if the scheduler decides to run some other user task. In this case, it will have different user space mappings, and thus the page table needs to be changed.

Point 3.3.2

Because the kernel's virtual address space is the same for all kernel threads, there is often no need to switch the page table upon a context switch. For example, while switching from user mode to kernel mode, there is no need to switch the page table. A need only arises when we are running a new user task, where the user space mappings change.

Thread Context Switch

As we have discussed earlier, for the Linux kernel, especially its scheduler, the basic unit of scheduling is a *task*, which is a single thread. It is true that the kernel supports the notion of a thread group, however all major scheduling decisions are taken at the level of tasks, i.e., single threads of execution.

> **Point 3.3.3**
>
> A task is the atomic unit of scheduling in the kernel.

Let us now come to the problem of switching between threads that belong to the same thread group. This should, in principle, be a more lightweight mechanism than switching between unrelated processes. There should be a way to optimize this process from the point of view of performance and total effort. The Linux kernel supports this notion.

Up till now we have maintained that each thread has its dedicated stack and TLS region. Here, TLS stands for *Thread Local Storage.* It is a private storage area for each thread. Given that we do not want to flush the TLB or switch the page table, we can do something very interesting. We can mandate all threads to actually use <u>the same</u> virtual address space like kernel threads. This is a reasonable decision for all memory regions other than the stack and the TLS region. Here, we can adopt the same solution as kernel threads and pure-user threads (see Section 3.2.3). We simply use the same virtual address space and assign different stack pointers to different stacks. This means that all the stacks are stored in the same virtual address space. They are just stored in different regions. We just have to ensure that the spacing between them is large enough in the virtual address space to ensure that one stack does not overflow and overwrite the contents of another stack. If this is done, then we can nicely fit all the stacks in the same virtual address space. The same can be done for TLS regions. On an x86 machine that supports segmentation, doing this is even easier. We just set the value of the stack segment register to the starting address of the stack – it is a function of the id of the currently executing thread in the thread group. This design decision solves a lot of problems for us. There is no need to frequently replace the contents of the CR3 register, which stores the starting address of the page table. On x86 machines, any update to the CR3 register typically flushes the TLB also. Both are very expensive operations, which in this case are fortunately avoided.

> **Point 3.3.4**
>
> In Linux, different threads in a thread group share the *complete* virtual address space. The stack and TLS regions of the constituent threads are stored at different points in this shared space.

There is however a need to store and restore the register state. This includes the contents of all the general-purpose registers, privileged registers, the program counter and the ALU flags. Finally, we need to set the `current` pointer to the `task_struct` of the new thread.

To summarize, this is a reasonably lightweight mechanism. Hence, many kernels typically give preference to another thread from the same thread group as opposed to an unrelated thread.

Interrupt Context Switch

Whenever a HW interrupt from a device arrives, we need to process it quickly. Servicing a timer interrupt is often a routine matter, however, other interrupts especially non-maskable interrupts have much higher priorities. Moreover, interrupt handlers are special because of the restrictions placed on them in terms of their code size, need to access native hardware and the fact that they are independent of any user thread. They are also not allowed to use locks. Nevertheless, the same trick of reusing the user thread and making it a kernel thread is used. However, in this case, a dedicated interrupt stack is used. Recall that in Section 3.1.5, we had mentioned that we maintain a set of interrupt stacks per CPU. Whenever an interrupt arrives, we find a free stack and assign it to the current thread.

Interrupt handling in Linux follows the classical top half and bottom half paradigm. Here, the interrupt handler, which is known as the *top half*, does basic interrupt processing. However, it may be the case that a lot more work is required. This work is deferred to a later time, and is assigned to a lower-priority thread, which is classically referred to as the *bottom half*. Of course, this mechanism has become more sophisticated now; however, the basic idea is still the same: do the high-priority work immediately and defer the low-priority work to a later point in time. The bottom-half thread does not have the same restrictions that the top-half thread has. It thus has access to a wider array of features. Here also, the interrupt handler's (top-half's) code and variables are in a different part of the virtual address space (not in a region that is accessible to any user process). Hence, there is no need to flush the TLB or reload any page table. This speeds up the context switch process.

3.3.3 Details of the Context Switch Process

Let us explain the details of the context switch process. We shall focus on system call handlers. Interrupt and exception handlers work similarly. There are minor differences, which we shall point out. First, understand that whenever there is an event of interest (system call, interrupt or exception), the hardware does a minimal amount of context saving and transfers control to the relevant handler. At this point, there is an automatic mode change (from user mode to kernel mode).

System Call Handlers

The context switch process is very architecture-specific. This typically involves a fair amount of hardware support and the code needs to be written at the assembly level. Recall that we need to explicitly store registers, ALU flags and other machine-specific information. The code for effecting a context switch for the x86-64 architecture can be found in arch/x86/entry/entry_64.S. The job of the functions and macros defined in this assembly program is to store the context of the current thread. For system calls, there is a common entry point on all 64-bit x86 machines. It is the entry_syscall_64 function. It is defined using the SYM_CODE_START directive. This directive indicates that the function is written in assembly language. Assembly language is needed because we access individual registers, especially many privileged registers, which are known as model-specific registers (MSR registers) in the x86-64 architecture.

Let us now look in detail at the steps involved in saving the context after a system call is made using the *syscall* instruction. The initial steps are performed automatically by hardware, and the later steps are performed by the system call handler. Note that during the process of saving the state, interrupts are often disabled. This is because this is a very sensitive operation, and we do not want to be interrupted in the middle. If we allow interruptions, then the state will be partially saved and the rest of the state will get lost. Hence, to keep things simple it is best to disable interrupts at the beginning of this process and enable them when the context is fully saved. Of course, this does delay interrupt processing a little bit; however, we can be sure that the context was saved correctly. Let us now look at the steps.

1. The hardware stores the program counter (`rip` register) in the register `rcx` and stores the flags register `rflags` in `r11`. Before making a system call, it is assumed that the two general purpose registers `rcx` and `r11` do not contain any useful data.

2. However, if there is an interrupt, then we cannot afford this luxury because interrupts can arrive at any point of time. In this case, the hardware needs to use MSR registers and dedicated memory regions to store the state. Specifically, we need to be concerned about storing the values of `rip` (PC), `CS` (code segment register) and `rflags`. These registers are ephemeral and change instruction to instruction. On many x86 machines, the hardware pushes them on to the current stack. This means that the hardware needs to read the value of the stack pointer and update it as well.

3. Subsequently, the software code of the interrupt handler takes over. It invokes the `swapgs` instruction to store the contents of the `gs` segment register in a pre-specified address (stored in an MSR). Recall that the `gs` segment plays a vital role in maintaining information regarding the current task – it stores the start of the per-CPU region.

4. Almost all x86 and x86-64 processors define a special segment in each CPU known as the Task State Segment or TSS. The size of the TSS segment is small, but it is used to store important information regarding the context switch process. It was previously used to store the entire context of the task. However, these days it is used to store a part of the overall hardware context of a running task. On x86-64 machines, the stack pointer (`rsp`) is stored on it. There is sadly no other choice. We cannot use the kernel stack because for that we need to update the stack pointer – the old value will get lost. We also cannot use a general-purpose register. Hence, a separate memory region such as the TSS segment is necessary.

5. Finally, the stack of the current process can be set to the kernel stack.

6. We can now push the rest of the state to the kernel stack. This will include the following:

   (a) The data segment register
   (b) The stack pointer (get `rsp` from the TSS)

    (c) `r11` (flags)

    (d) The code segment register

    (e) `rcx` (program counter)

    (f) The rest of the general-purpose registers

To restore the state, we need to exactly follow the reverse sequence of steps.

`sysret` and `iret` Instructions

The `sysret` instruction is used to return from a system call. It transfers the contents of `rcx` (saved instruction pointer) to `rip` and `r11` to `rflags`. For the entire sequence of instructions, we cannot disable interrupts – the slowdowns will be prohibitive. It is possible that an interrupt arrives between restoring the stack pointer (`rsp`) and executing `sysret`. At some of these points, it is possible to execute an interrupt handler using its dedicated stack (from the interrupt stack table). There will be no correctness issue.

    The `iret` instruction is used to return from interrupts. Specifically, it restores the values of `rip`, the code segment register and `rflags` from the stack. Note that `rip` is set at the end of this process. Setting the instruction pointer is tantamount to returning from the interrupt. Given that this instruction pointer points to a program counter in the virtual address space of the user process, we are effectively jumping to the return address in the user process.

    Finally, note that both of these instructions cause a mode change: kernel mode to user mode.

Additional Context

Along with the conventional hardware context, there are additional parts of the hardware context that need to be stored and restored. Because the size of the kernel stack is limited, it is not possible to store a lot of information there. Hence, a dedicated structure called a `thread_struct` is defined to store all extra and miscellaneous information. It is defined at the following link: arch/x86/include/asm/processor.h.

    Every thread has TLS regions (thread local storage). It stores variables specific to a thread. The `thread_struct` stores a list of such TLS regions (starting address and size of each), the stack pointer (optionally), the segment registers (`ds`,`es`,`fs` and `gs`), I/O permissions and the state of the floating-point unit.

3.3.4 Context Switch Process: Kernel Code

Once the context is fully saved, the user thread starts to execute in "kernel mode". It can then service the interrupt or system call. Once this is done, the thread needs to check if there is additional work to do. It checks if there is a high-priority user thread that is waiting. In such a case, that other high-priority thread should run as opposed to the erstwhile user thread continuing.

    The kernel thread calls `exit_to_user_mode_loop` in kernel/entry/common.c, whose job is to basically check if there is other high-priority work to be done. If there is work, then there is a need to call the scheduler's `schedule` function. It finds the task to run next and effects a context switch.

The `context_switch(runqueue, prev_task, next_task)` function is invoked. It is defined in kernel/sched/core.c. It takes as input the runqueue, which contains all the ready processes, the previous task and the next task that needs to run. There are five major steps in the context switch process.

- `prepare_task_switch`: prepare the context switch process

- `arch_start_context_switch`: initialize the architectural state (if required). At the moment, x86 architectures do basic sanity checks in this stage.

- Manage the `mm_struct` structures (memory maps) for the previous and next tasks

- `switch_to`: switch the register state and stack

- `finish_task_switch`: finish the process

Prepare the Task Switch

There are two tasks here, *prev* and *next*. *prev* was running and *next* is going to run. If they are different tasks, we need to set the status of the *prev* task as "not running".

Switch the Memory Structures

Every `task_struct` has a member called `struct mm_struct *mm`, as we have seen before. It contains a pointer to the page table and a list of VMA (virtual memory) regions. The `task_struct` also has a member called `struct mm_struct* active_mm`, which has a special role.

There are two kinds of kernel threads. One kind are user threads that have been temporarily converted to kernel threads after a system call or interrupt. The other type of kernel threads are pure kernel threads that are not associated with any user-level threads. For a user-level thread, `mm` and `active_mm` are the same. However, for a kernel-level thread, `mm` is set to NULL and `active_mm` points to the `mm` of the last user process. The reason that we maintain this state is because even if a pure kernel thread is executing, it should still have a reference to the last user process that was running on the CPU. In case, there is a need to access the memory of that user process, it should be possible to do so. Its mappings will be alive in the TLB. This is a performance-enhancing measure.

Listing 3.11: Code for switching the memory structures (partial code shown with adaptations)
source : kernel/sched/core.c#L5266

```
if (! next ->mm) {
    next ->active_mm = prev ->active_mm ;
    if (prev ->mm) {
        // increment reference count
        mmgrab (prev ->active_mm);
    } else {
        prev ->active_mm = NULL;
    }
} else {
```

```
    ...
    if (!prev->mm) {
        prev->active_mm = NULL;
    }
}
```

Some relevant code is shown in Listing 3.11. If `next->mm` is NULL, it means that we are switching to a kernel thread. In this case, we simply set the `active_mm` of the kernel thread to that of the previous thread. This means that we are just transferring the `active_mm`, which is the `mm_struct` of the last user process that executed. If the previous thread was a user thread, then we increment its reference count. Otherwise, we set the `active_mm` field of the previous thread to NULL because this information is not required any more.

Consider the other case. Assume a switch to a user process: `next->mm` is not NULL. First, we compare `prev->active_mm` and `next->mm`. If both are the same, then it means that the user process that last executed on the CPU is going to execute again. There could be a lot of kernel threads that have executed in the middle, but finally the same user process is coming back. Since its page table and TLB state have been maintained, there is no need to flush the TLB. This improves performance significantly. Specifically, if `prev->mm` is NULL, it means that the previous process is a kernel thread. Given that the current process is a user process, there is no need for the kernel thread to maintain its `active_mm` pointer. It is set to NULL.

Switching the Registers and the Stack

The `__switch_to` function accomplishes this task by executing the steps to save the context in the reverse order (context restore process). The first step is to extract all the information in the `thread_struct` structures and restore them. They are not very critical to the execution and thus can be restored first. Then the thread local state and segment registers other than the code segment register are restored. Finally, the `current` task pointer, a few of the registers and the stack pointer are restored.

Finishing the Process

The function `finish_task_switch` completes the process. It updates the process states of the *prev* and *next* tasks and also updates the timing information associated with the respective tasks. This information is used by the scheduler. Sometimes it can happen that the kernel uses more memory than the size of its virtual address space. On 32-bit systems, the kernel can use only 1 GB. However, there are times when it may need more memory. In this case, it is necessary to *temporarily* map some pages to kernel memory (known as *kmap* in Linux). These pages are typically unmapped in this function before returning back to the user process.

Finally, we are ready to start the new task !!! We set the values of the rest of the flags, registers, the code segment register and finally the instruction pointer.

Trivia 3.3.1

One will often find statements of this form in the kernel code:

```
if (likely (<some condition >)) {...}
if (unlikely (<some condition >)) {...}
```

These are hints to the branch predictor of the CPU. The term *likely* means that the branch is most likely to be taken, and the term *unlikely* means that the branch is most likely to be not taken. These hints increase the branch predictor accuracy, which is vital for good performance.

Trivia 3.3.2

One often finds statements of the form:

```
static __latent_entropy struct task_struct *
    copy_process (...){...}
```

Here, we are using the value of the `task_struct*` pointer as a source of randomness. Many such random sources are combined in the kernel to create a good random number source that can be used to generate cryptographic keys.

3.4 Summary and Further Reading

3.4.1 Summary

Summary 3.4.1

1. A *process* is a program in execution.

2. `struct task_struct` is the key data structure that stores all the information related to processes.

3. The pointer to the current `task_struct` is given by the `current` macro. It is stored in a per-CPU storage area, which is pointed to by the `gs` segment register.

4. The task states in Linux are TASK_RUNNING (ready to execute or currently executing), TASK_ZOMBIE (completed), TASK_STOPPED (suspended due to a `SIGSTOP` signal), TASK_INTERRUPTIBLE and TASK_UNINTERRUPTIBLE. The last two states are blocked states.

5. Once a task completes, it calls the `exit` system call. Subsequently, it enters the ZOMBIE state. The `SIGCHLD` signal is sent to the parent. It subsequently calls the `wait` system call and collects the exit status of the child. Subsequently, the child's state is fully erased.

6. Each task has a kernel stack whose size is limited to 8 KB.

7. Linux tasks have 140 priorities. It has 100 real-time priorities (increasing from 0 to 99) and 40 user-level priorities (100-139). The default process priority is 120, which can be changed using the `nice` command (varies from -20 to 19).

8. The key structure in a process for storing the list of virtual memory regions (`struct vma`) and the page table is `struct mm_struct`. The set of VM regions are stored using a maple tree: a B+ tree with variable branching factors (different per level).

9. Every process has an id known as its *pid*, which is a number that uniquely identifies it in its namespace. A namespace is an isolated set of processes. All the processes in a namespace can be suspended, checkpointed and migrated to a new machine. They can then be seamlessly resumed on the new machine.

10. Namespaces are arranged hierarchically. Every process is also visible to all of its ancestral namespaces. It can be referenced by different *pid* numbers in different namespaces. All of them however point to the same `struct pid`, which is a structure that contains all the details of the process.

11. Processes themselves are organized in a tree-like structure. The root of this tree is the *init* process ($pid = 1$) for all user space processes. Similarly, kernel threads also have a hierarchical structure where every kernel task has a parent. The root of this tree is the *kthreadd* process ($pid = 2$).

12. A new process is created using the forking mechanism. Here, a child process is created by fully copying the memory and execution state of the parent process. The child and the parent are separate entities; however, the child is initialized with a copy of the memory map of the parent.

13. The copy-on-write mechanism is used to ensure that the child has a separate physical address space. Whenever there is a write to a page that is shared between the child and parent process, a new copy is created for the writer. The writes are directed to the new copy of the physical page (frame). Note that there is a need to update the page table and the TLB contents of the writer process.

14. The child process can decide to go its own way and totally replace its memory image with that of another binary. It can do so by calling the `exec` family of system calls. They completely clean up the memory image of the child process and load the memory image corresponding to the binary specified as the argument of the `exec` call. The program counter is initialized to the first instruction of the text section. The child process starts executing the new binary from the beginning.

15. The hardware context comprises the values of all the registers (general-purpose and privileged), the next program counter and the ALU flags. This context needs to be saved and later restored.

16. All the kernel threads share the kernel virtual address space. This insight can be used to eliminate TLB flushes and page table switches whenever there is a context switch to kernel mode or there is a context switch between kernel threads.

    (a) Split the virtual space between the user space and kernel space.

    (b) The mappings for all the kernel pages across all the processes (user and kernel) are identical.

    (c) There is no need to flush the TLB when there is a kernel to user-mode transition and the interrupted user process is being resumed once again.

    (d) A need to arises for a TLB flush and a page table switch only when we are resuming a different user process.

17. In practice, it is a wise idea to share the complete virtual address space across all the threads in a thread group. The illusion of separate stacks can be provided by storing the stacks of different threads at different points in the virtual address space and by ensuring that the stacks of two threads never overlap. The same can be done for thread-local regions.

18. The process of storing the context is a very elaborate sequence of steps where there is a need to create bespoke solutions for storing the next PC, flags, segment registers, the stack pointer, MSRs and other general-purpose registers. Restoring the context follows exactly the reverse sequence of steps.

3.4.2   Further Reading

Readers should start by looking at the man (manual) pages of the following system calls: `fork`, `clone`, `exec`, `exit` and `wait`. This will give them a feel of how processes are created, destroyed and managed in Linux. They will appreciate the user space API that interacts with the kernel to manage processes.

   The next activity will be to use kernel tracing tools to understand the activity of processes. Some of the tools in this space are *ftrace* (traces function calls and events), *perf* (performance monitoring), *systemtap* (dynamic instrumentation of the kernel and user processes) and *eBPF* (observability and sandbox API). These tools help a process monitor the activity in another user process or even the kernel. They can also attach themselves to two probing mechanisms: *kprobes* and *uprobes*. *kprobes* inserts a breakpoint at a given address in the kernel code. Whenever that address is reached, a call is made to the *kprobes* handler function. It can be used to inspect the state of the kernel and debug the execution from then on. The *uprobes* mechanism does the same for user space processes. These

are dynamic mechanisms that allow breakpoints to be placed at run time.

On the other hand, the *tracepoints* mechanism in the kernel is a static mechanism. The *trace points* are statically placed instrumentation points in the kernel code. They need to be there at compile time. Whenever they are reached, the corresponding handler is called. Relevant data can be collected, and it can also be made available to other higher level tools such as *ftrace*.

In the world of kernel-level code instrumentation (code modification) and debugging, *SystemTap* and Berkeley packet filters (BPF and eBPF) have a very special place. SystemTap is a high-level wrapper on `kprobes` and `uprobes`. The user specifies the functions and locations within functions that need to be tapped into. Subsequently, probes are inserted at those points. It is possible to analyze the state at these probe points.

Extended Berkeley Packet Filter (eBPF) allows for the creation of a sandbox – a restricted environment for running a small set of processes that do some sort of performance monitoring, safety checking and network packet processing. eBPF programs can be written in C. They are compiled to custom byte code (an intermediate architecture-independent representation). The Linux kernel uses a small custom virtual machine to run such code. The byte code is compiled at run time using a JIT (just-in-time) compiler and is also checked. The reader is encouraged to write such programs and connect them to the kernel and user processes such that they can perform the following tasks: tracing and profiling kernel code, detecting malicious activity and enforcing security policies by monitoring the system calls made by the process.

Exercises

Ex. 1 — Why do we need a kernel stack in a multiprocessor operating system? Explain with an example.

Ex. 2 — Why do we use the term "kernel thread" as opposed to "kernel process"?

Ex. 3 — How does placing a limit on the kernel thread stack size make kernel memory management easy?

Ex. 4 — If the kernel wants to access physical memory directly, how does it do so using the conventional virtual memory mechanism?

Ex. 5 — Explain the design and operation of the kernel linked list structure in detail.

Ex. 6 — Why cannot the kernel code use the same user-level stack and delete its contents before a context switch?

Ex. 7 — What are the advantages of creating a child process with *fork* and *exec*, as compared to a hypothetical mechanism that can directly create a pro-

cess given the path to the binary?

Ex. 8 — Assume that there are some pages in a process such as code pages that need to be read-only all the time. How do we ensure that this holds during the forking process as well? How do we ensure that the copy-on-write mechanism does not convert these pages to "non-read-only"?

Ex. 9 — What is the role of the TSS segment in the overall context switch process?

Ex. 10 — Do we need extra registers for servicing a hardware interrupt? Are the existing set of general purpose registers enough for implementing an interrupt handler? Explain your answer.

Ex. 11 — What is the role of the `active_mm` field?

Ex. 12 — What makes `rip`, `rsp`, `rflags` and the code segment register special?

Ex. 13 — Why do we use registers to store the values of `rip` and `rflags` in the case of system calls, whereas we use the stack for interrupts?

* Ex. 14 — To save the context of a program, we need to read all of its registers, and store them in the kernel's memory space. The role of the interrupt handler is to do this by sequentially transferring the values of registers to kernel memory. Sadly, the interrupt handler needs to access registers for its own execution. We thus run the risk of inadvertently overwriting the context of the original program, specifically the values that it saved in the registers. How do we stop this from happening?

Ex. 15 — How does the design of a namespace facilitate its migration?

Ex. 16 — Consider a situation where a process exits, yet a few threads of that process are still running. Will those threads continue to run? Explain briefly.

Ex. 17 — Why are `idr` trees used to store *pid* structures? Why can't we use BSTs, B-Trees, and hash tables? Why is it effective?

Ex. 18 — Which two trees does the `idr` tree combine? How and why?

Ex. 19 — What is the need of a `struct pid` in addition to a `pid` number?

Ex. 20 — What are the advantages of dynamically loaded libraries? How do they save memory space (at runtime)?

Ex. 21 — Describe the operation of a dynamic loading library (DLL). Focus on the following issues.

   a)What happens if multiple programs need to use the same DLL concurrently?

   b)How do we manage two versions of the same DLL?

   c)Assume that the location of the DLL changes across two versions of the same operating system. Will programs stop working?

d)How do DLLs support global and static variables? If the same DLL is being used concurrently, wouldn't this cause a problem?

Ex. 22 — How do we implement a `clone` system call where a part of a process's memory map is copied, and the rest is shared? This can, for instance, be used to create a new thread that has a separate stack but shares the heap.

Ex. 23 — How do we create a thread-local storage area?

* **Ex. 24** — Consider a thread library such as *pthreads*. Here, we create a new thread and assign it with a function to execute. Once the function finishes execution, the thread is supposed to be destroyed. However, the return value of the function needs to be preserved and made available to the parent thread. How is such a mechanism implemented?

Ex. 25 — How are the radix and augmented trees combined? What is the need for combining them? Answer the latter question in the context of process management.

Open-Ended Questions

Ex. 26 — What is the difference between a shell and a terminal?

Ex. 27 — Read about the following Linux commands and explain their operation: `objdump`, `nm`, `strip`, `ld` and `ldd`.

Chapter 4

System Calls, Interrupts, Exceptions and Signals

In this chapter, we will delve into the details of system calls, interrupts, exceptions and signals. The first three are the only methods to invoke the OS kernel, or in other words bring something to its attention. It is important to bear in mind that the kernel code normally remains dormant. It comes into action only after three *events of interest*: system calls, interrupts and exceptions. In a generic context, all three of these events are often referred to as *interrupts*. They involve transferring control from one process to a dedicated kernel handler. Note that sometimes specific distinctions are made such as using the terms "hardware interrupts" and "software interrupts". Hardware interrupts refer to classical interrupts generated by I/O devices whereas software interrupts refer to system calls and exceptions.

The classical method of making system calls on x86 machines is to invoke the instruction `int 0x80` that simply generates an interrupt with interrupt code 0x80. The generic interrupt processing mechanism is used to process the system call. Modern machines have the `syscall` instruction, which is more direct and specialized (as we have seen in Section 3.3.3), even though the basic mechanism is still the same. x86 processor further simplify things. They treat exceptions as a special type of interrupts. It is a good idea to group events in this fashion – kernel routines can be reused.

All hardware interrupts have their own interrupt codes – they are also known as *interrupt vectors*. Similarly, all exceptions have their unique codes and so do system calls. Whenever any such event of interest happens, the hardware first determines its type. Subsequently, the corresponding table with the code of the event of interest is accessed. For example, an interrupt vector is used to index interrupt handler tables. Each entry of this table points to a function that is meant to handle the interrupt.

Finally, we shall discuss communication in the reverse direction. Sometimes the kernel needs to inform user processes about some event that they may be interested in. A user process starts with registering a function pointer with some kernel-specific event, which is known as a *signal*. Whenever the kernel needs to raise a signal, it invokes this function in the user process's context.

Such a function is known as a signal handler and plays the role of an interrupt handler, albeit in the context of a regular user process. This mechanism is very useful in applications with graphical user interfaces (GUIs). Whenever there is a mouse click, the kernel needs to inform the foreground application about it. This is easily achieved using signal handlers. These signal handlers (also referred to as event handlers) implement application-specific logic based on the details associated with the mouse click.

In this chapter, we shall realize that bidirectional communication between user applications and the kernel is very important. All user processes require OS services and thus elaborate mechanisms need to be provided. Similarly, there needs to be a mechanism for the kernel to inform user applications about specific events. Beyond this simple explanation, there lies a lot of detail, which we shall appreciate in this chapter.

Organization of this Chapter
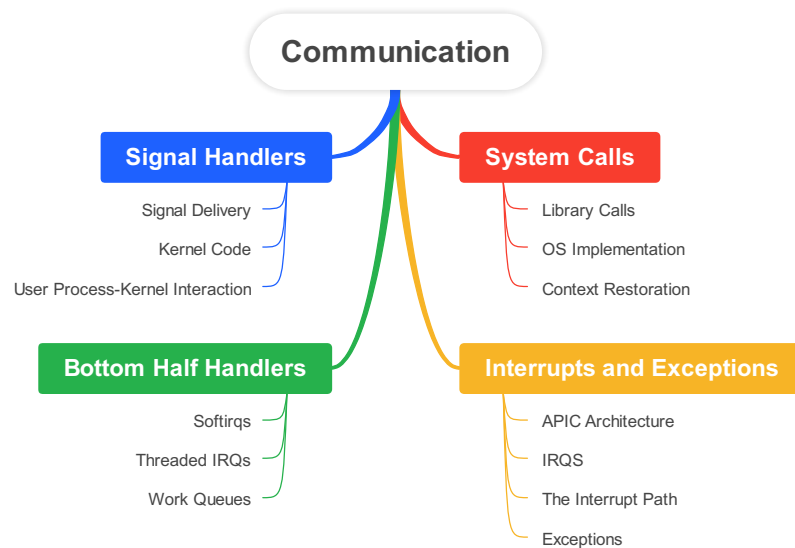
Figure 4.1: Organization of this Chapter

Figure 4.1 shows the organization of this chapter. We will start with an in-depth study of system calls. It is necessary to understand the life cycle of a library call and understand how it prepares the arguments for a system call. We shall see that after several stages of processing, the system call is finally made. There are precise rules for making system calls. It is associated with detailed register- and stack-usage semantics. The system call ultimately ends up calling a large number of kernel routines. We shall end this section with providing a few more details about running the scheduler, storing and restoring the context.

Next, we shall study the interrupt architecture on x86 machines. The CPU relies on a bunch of chips in the chipset known as advanced programmable

interrupt controllers (APICs). These APICs are hierarchically organized and can perform complex interrupt processing. Ultimately, after passing through a sequence of APICs, interrupts arrive at the CPU. The correct interrupt handler is invoked. However, servicing the interrupt is not as easy as directly jumping to the corresponding logic in the device driver code. We shall appreciate the fact that even identifying the device that led to an interrupt is quite complicated. Exceptions are also handled like interrupts on x86 machines. They broadly follow the same processing path.

Interrupt handlers are ultra-high-priority tasks. They are outside the normal scheduling regime. This means that no process is scheduled if an interrupt handler needs to run. As a result, there are a lot of restrictions on interrupt handlers. They cannot make blocking calls and cannot access certain subsystems of the kernel such as accessing user space memory or making kernel-level `malloc` calls. Their execution duration should also be short. Hence, there is a need to defer some work that can be done at a later point of time. Such handlers are known as bottom-half handlers that finish the leftover work of regular interrupt handlers (known as top-half handlers). We shall discuss three types of bottom-half handlers: softirqs, threaded IRQs and work queues. They have different trade-offs in terms of their priority and the type of actions that they are allowed to do.

Finally, we shall discuss the signal subsystem in Linux. In this subsection, we first introduce C programming constructs that are used to register signal handlers and handle signals. Then we shall delve into the intricacies of the corresponding library and kernel code. The focus will be on the interaction of user processes and the kernel, steps involved in entering the signal context (one that runs the signal handler) and restoring the state of the user process that was interrupted.

4.1   System Calls

4.1.1   Life of a Library Call

Consider the simple piece of C code shown in Listing 4.1. It shows a call to the *printf* library call (part of the standard C library). It prints the string "Hello World" to the terminal. Recall that a library call encapsulates a system call. It prepares the arguments for the system call, sets up the environment, makes the system call and then appropriately processes the return values. The *glibc* library on Linux contains all the relevant library code for the standard C library.

Listing 4.1: Example code with the *printf* library call

```
#include <stdio.h>

int main() {
    printf ("Hello World \n");
}
```

Let us now understand this process in some detail. The signature of the *printf* function is as follows: `int printf(const char* format, ...)`. The `format` string is of the form ``The result is %d, %s''. It is succeeded by a sequence of arguments, which replace the format specifiers (``%d'' and ``%s'')

in the format string. The ellipses ... indicate that the number of arguments is variable.

A sequence of functions is called in the *glibc* code. The sequence is as follows: printf → __printf → vfprintf → printf_positional → outstring → PUT. Gradually the signature changes – it becomes more and more generic. This ensures that other calls like fprintf that write to a file are all covered by the same function as special cases. Note that Linux treats every device as a *file* including the terminal. The terminal is a special kind of file, which is referred to as *stdout*. The function vfprintf accepts a generic file as an argument, which it can write to. This *generic file* can be a regular file in the file system or the terminal (*stdout*). The signature of vprintf is as follows:

```
int vfprintf (FILE *s, const CHAR_T *format, va_list ap,
    unsigned int mode_flags);
```

Note the generic file argument FILE *s, the format string, the list of arguments and the flags that specify the nature of the I/O operation. Every subsequent call generalizes the function further. Ultimately, the control reaches the new_do_write function in the *glibc* code (fileops.c). It makes the write system call, which finally transfers control to the OS. At this point, it is important to digress and make a quick point about the generic principles underlying library design.
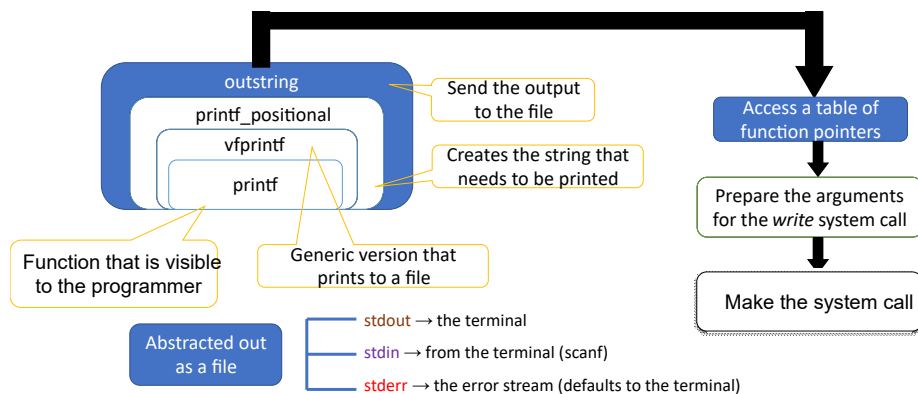
Figure 4.2: General concepts underlying library design

General Concepts in Library Design

Figure 4.2 shows the generic design of the *printf* library call. The *printf* function is visible to the programmer. It, by default, writes to the *stdout* (standard out) file (the terminal). It is the default/standard output stream for all programs.

Let us quickly mention the two other standard streams recognized by the *glibc* library. The first is the standard error stream (*stderr*). *stderr* is normally mapped to the terminal, however this mapping can be changed. Note that there is another standard file defined – *stdin* – which is the standard input stream. Whenever we call the *scanf* function in C, the *stdin* input stream is used to read input from the terminal.

Figure 4.2 shows the sequence of calls that are made. They have different levels of abstraction. The `vfprintf` function is more generic. It can write to any file including *stdout*. The `printf_positional` function creates the string that needs to be printed. It sends the output to the `outstring` function that ultimately dispatches the string to the function that writes to the file. The file write is achieved by the *write* system call, which sends the string that needs to be printed along with other details to the kernel.

4.1.2   The OS Side of Things

There are two ways to make a system call in Linux. We can either use the older method, which is to issue a software interrupt `int 0x80` or call the *syscall* instruction. Regardless of the method used, we arrive at the entry point of a system call, which is the `do_syscall_64` function defined in arch/x86/entry/entry_64.S. At this point, there is a ring level switch and interrupts are switched off. The reason to turn off interrupts is to ensure that the context is saved correctly. If there is an interrupt in the middle of saving the context, there is a possibility that an error may be induced due to race conditions. Hence, the context saving process cannot terminate prematurely. Saving the context is a short process and masking interrupts during this process normally does not create a lot of performance issues in handling critical tasks. Interrupts can be enabled as soon as the context is saved.

Linux has a standard system call format. It is shown in Table 4.1 that shows which register stores which type of argument. For instance, `rax` stores the system call number. Six more arguments can be supplied via registers as shown in Table 4.1. If there are more arguments, then they need to be transferred via the user stack. The kernel can read user memory, and thus it can easily retrieve these arguments. However, passing arguments using the stack is not the preferred method. It is a much slower method as compared to passing values via registers.

Note that a system call is a *planned activity*, as opposed to an interrupt. Hence, we can keep some registers free such as `rcx` and `r11` by spilling their contents to the stack. Recall that the PC (of the return address) and the flags are automatically stored in these registers once a system call is made. The system call handler subsequently stores the contents of these registers on the kernel stack.

Attribute	Register
System call number	`rax`
Arg. 1	`rdi`
Arg. 2	`rsi`
Arg. 3	`rdx`
Arg. 4	`r10`
Arg. 5	`r8`
Arg. 6	`r9`

Table 4.1: Convention for system call arguments

Let us now discuss the `do_syscall_64` function more. After basic context

saving, interrupts are enabled, and then the function accesses a system call table as shown in Table 4.2. Given a system call number, the table lists the pointer to the function that handles the specific type of system call. This function is subsequently invoked. For instance, the *write* system call ultimately gets handled by the *ksys_write* function, where all the arguments are processed, and the real work is done.

Number	System call	Function
0	read	sys_read
1	write	sys_write
2	open	sys_open
3	close	sys_close
4	stat	sys_newstat
5	fstat	sys_newfstat
6	lstat	sys_newlstat
7	poll	sys_poll
8	lseek	sys_lseek
9	mmap	sys_mmap
10	mprotect	sys_mprotect
11	munmap	sys_munmap

Table 4.2: Entries in the syscall table

4.1.3   Returning from a System Call

The kernel is sadly not a very grateful friend. Once a process goes to the kernel, there is no guarantee that it will immediately get scheduled once the work of the system call is done. The kernel can decide to do its own work such as perform routine bookkeeping, update its data structures or service devices by running kernel threads. It can also schedule other user processes.

The kernel starts out by checking the TIF_NEED_RESCHED bit in the flags stored in the thread_info structure (accessible via task_struct). This flag is set by the scheduler when it feels that the current task has executed for a long time, and it needs to give way to other processes or there are other higher priority processes that are waiting. Sometimes threads explicitly request for getting preempted such that other threads get a chance to execute (via sched_yield). In this case, the thread that wishes to yield the CPU gets the TIF_NEED_RESCHED bit set.

If this flag is set, the scheduler needs to run and find the most worthy task (user process or kernel thread) to run next. It uses complex algorithms to find the next task. Note that it treats the TIF_NEED_RESCHED bit as a directive to run the scheduler. Once the scheduler runs, it makes its independent decision. It may decide to continue with the same task, or it may decide to start a new task on the same core. This is purely the scheduler's prerogative.

After a task is chosen, its context needs to be restored. The context restore mechanism follows the reverse sequence vis-á-vis the context switch process. The issue of segment registers needs to be discussed here. On x86-64, the ds, es and ss segment registers are typically not used. Hence, the need to save and

restore them is often not present. However, the `fs` and `gs` segment registers are used. `fs` stores a pointer to the thread local storage area (TLS) and `gs` stores a pointer to per-CPU data structures. Hence, they are stored and restored as regular registers – they are a part of a task's context. The code segment register `cs` register is special. All kernel threads (running in ring 0) typically use the same value for `cs` and so do all user processes. Hence, whenever there is a ring level switch, the value of `cs` can be automatically inferred by hardware and appropriately set.

Finally, the kernel calls the *sysret* instruction that sets the value of the PC and completes the control transfer back to the user process. It also changes the ring level or in other words effects a mode switch (from kernel mode to user mode).

4.2   Interrupts and Exceptions

Figure 4.3 shows the structure of the Interrupt Descriptor Table (IDT) that is pointed to by the `idtr` register. As we can see, regardless of the source of the interrupt, ultimately an integer code called an *interrupt vector* gets associated with it. It is the job of the hardware to assign the correct interrupt vector to an interrupting event. Once this is done, a hardware circuit accesses the IDT using the interrupt vector as the index.
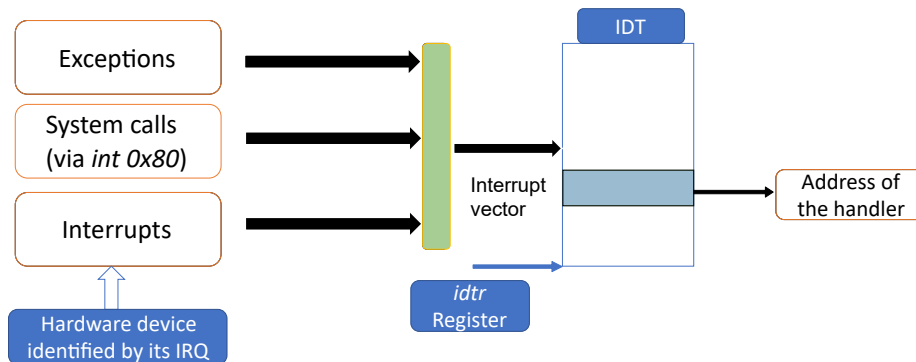
Figure 4.3: The Interrupt Descriptor Table (IDT)

Accessing the IDT is a simple process. A small module in hardware simply finds the starting address of the IDT by reading the contents of the `idtr` register and then accesses the relevant entry using the interrupt vector. The output is the address of the interrupt handler, whose code is subsequently loaded. The handler finishes the rest of the context switch process and begins to execute the code to process the interrupt. Let us now understand the details of the different types of handlers.

Intel processors have APIC (Advanced Programmable Interrupt Controller) chips that do the job of liaising with hardware and generating interrupts. These dedicated chips are sometimes known as just interrupt controllers. There are two kinds of interrupt controllers on standard Intel machines: LAPIC (local APIC), a per-CPU interrupt controller, and the I/O APIC. There is only one

I/O APIC for the entire system. It manages all external I/O interrupts. Refer to Figure 4.4 for a pictorial explanation.
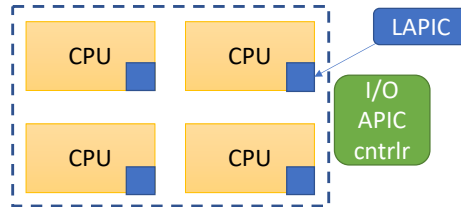
Figure 4.4: Interrupt processing mechanism in x86 processors

4.2.1 APICs

Figure 4.5 represents the flow of actions. We need to distinguish between two terms: interrupt request (IRQ) and interrupt number/vector. The interrupt number or interrupt vector is a unique identifier of the interrupt and is used to identify the interrupt service routine that needs to run whenever the interrupt is generated. The IDT is indexed by this number.

The interrupt request(IRQ), on the other hand, is a hardware signal that is sent to the interrupt controller indicating that a certain hardware unit's request needs to be serviced. A modern CPU has many IRQ lines (see Figure 4.5). For example, one line may be dedicated for the keyboard, one for the mouse, one for the mouse, and so on. In older systems, there was a one-to-one mapping between IRQ lines and interrupt vectors. However, with the advent of programmable interrupt controllers (read APICs), this has been made more flexible. The mappings can also be changed dynamically. It is possible for a single IRQ line to generate many types of interrupts with different interrupt vectors. For example, the network card can signal the completion of a request, or it can also indicate that there was an error in transmitting a message in an internal queue. Similarly, it is also possible to generate the same interrupt vector for different IRQ lines, although this situation is rare. In general, there is a many-to-many mapping, which is dynamically programmable. Note that it is the job of the LAPIC to generate interrupt vectors and send them to the CPU. Let us elaborate.

The flow of actions (for the LAPIC) is shown in Figure 4.5.

1. The first step is to check if interrupts are enabled or disabled. Recall that we discussed that often there are sensitive portions of the kernel's execution, where it is a wise idea to disable interrupts such that no correctness problems are introduced. Interrupts are typically not lost. They are queued in the hardware queue in the respective APIC and processed in priority order when interrupts are enabled back again. Of course, there is a possibility of overflows. This is a rare situation but can happen. In this case interrupts will be lost. In this context, let us differentiate between disabling and masking interrupts. They are different terms. Disabling interrupts is like a sledgehammer, where all the interrupts are temporarily disabled. However, masking is a more fine-grained action, where only certain interrupts are disabled in the APIC. Akin to disabling, the interrupts
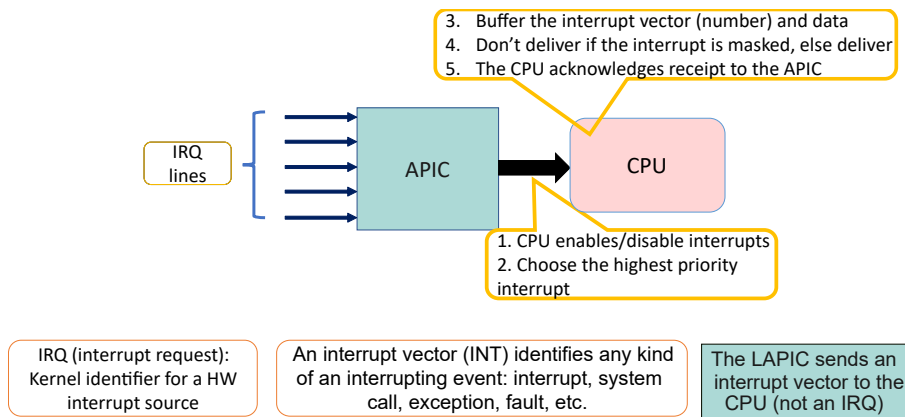
Figure 4.5: Interrupt processing flow

are queued in the APIC and presented to the CPU at a later point of time when they are unmasked.

2. Let us assume that interrupts are enabled. The LAPIC chooses the highest priority interrupt and finds the interrupt vector for it. In legacy systems, the voltage of a single IRQ line is raised from zero to one. The LAPIC simply maps the IRQ line to an interrupt vector based on an internal table. However, in modern MSI architectures that use message-signaled interrupts, some data can also be written to APIC registers by the device. This information is used to appropriately process the interrupt and send the right interrupt vector to the concerned CPU.

3. The LAPIC buffers the interrupt vector and data, and then checks if the interrupt is masked or not.

4. If it is masked, then it is added to a queue, otherwise it is delivered to the CPU.

5. The CPU needs to acknowledge that it has successfully received the interrupt and only then does the APIC remove the interrupt from its internal queue.

Let us now understand the roles of the different interrupt controllers in some more detail.

I/O APIC

There is only one I/O APIC chip in the entire system. It is not a part of any CPU core, instead it is typically a separate chip on the motherboard (part of the chipset). It maintains a redirection table, whose role is to receive interrupt requests from different devices, process them and dispatch the interrupts to the LAPICs. It is essentially an interrupt router. Many modern I/O APICs have 24 interrupt request lines. Typically, each device is assigned its IRQ number – the lower the number, higher the priority. A noteworthy mention is the timer interrupt, whose IRQ number is typically 0.

Local APIC (LAPIC)

Each LAPIC receives interrupts from the I/O APIC. It can also receive a special kind of interrupt known as an *inter-processor interrupts* (IPI) from other LAPICs. This type of interrupt is very important for kernel code. Assume that a kernel thread is running on CPU 5, and the kernel decides to preempt the task running on CPU 1. Currently, we are not aware of any method of doing so. The kernel thread only has control over the current CPU, which is CPU 5. It does not have any control over what is happening on CPU 1. The IPI mechanism is precisely designed to solve this problem. CPU 5 on the behest of the kernel thread running on it, can instruct its LAPIC to send an IPI to the LAPIC of CPU 1. This will be delivered to CPU 1, which will get interrupted. The usual set of actions will follow. It will switch its context and run the IPI interrupt handler on CPU 1. After doing the necessary bookkeeping steps, the kernel thread running on CPU 1 will realize that it was brought in because the kernel thread on CPU 5 wanted to replace the task running on CPU 1 with some other task. In this manner, one kernel thread can exercise its control over all CPUs. However, it does need the IPI mechanism to achieve this, which is hardware-based. Often, the timer chip is often housed inside the LAPIC. Depending upon the needs of the kernel, its interrupt frequency can be configured or even changed dynamically.

Distribution of Interrupts

The next question that we need to address is how are the interrupts distributed among the LAPICs? There are regular I/O interrupts, timer interrupts and IPIs. We can either have a static distribution or a dynamic distribution. In the static distribution, one specific core or a set of cores are assigned the role of processing a given interrupt. Of course, there is no flexibility when it comes to IPIs. Even in the case of timer interrupts, it is typically the case that each LAPIC generates periodic timer interrupts to interrupt its local core. However, this is not absolutely necessary, and some flexibility is provided. For instance, instead of generating periodic interrupts, it can be programmed to generate an interrupt at a specific point of time. In this case, this is a one-shot interrupt like an alarm – periodic interrupts are not generated. This behavior can be changed dynamically owing to the fact that LAPICs are programmable.

In the dynamic scheme, it is possible to send the interrupt to the core that is running the task with the least priority. This again requires hardware support. Every core on an Intel machine has a `task priority register`, where the kernel writes the priority of the current task that is executing on it. This information is used by the I/O APIC to deliver the interrupt to the core that is running the least priority process. This is a very efficient scheme, because it allows higher priority processes to run unhindered. If there are idle cores, then the situation is even better. They can be used to process all the I/O interrupts and sometimes even timer interrupts (if they can be rerouted to a different core).

4.2.2 IRQs

The file `/proc/interrupts` contains the details of all the IRQs and how they get processed (refer to Figure 4.3). Note that this file is relevant to only the author's machine and that too as of 2023.

The first column is the IRQ number. As we see, the timer interrupt is IRQ# 0. The next four columns show the count of timer interrupts received at each CPU. Note that there are many small values. This is because any modern machine has a variety of timers. The data is shown for the low-resolution LAPIC timer. In this case, a more high-resolution timer was used. Modern kernels prefer high-resolution timers because they can dynamically configure the interrupt interval based on the processes that are executing in the kernel. Many modern kernels are also tickless, which means that they have gotten away with periodic timer interrupts altogether. The term "2-edge" means that this is an edge-triggered interrupt on IRQ line 2. An astute reader will note that some interrupt remapping is happening. The interrupt was triggered on IRQ line 0; however, it got remapped to IRQ line 2 by the I/O APIC chip. From then on, it appears as if there has been an interrupt on IRQ #2. This behavior is programmable, and can definitely be used to coalesce interrupts from multiple sources. "edge" corresponds to edge-triggered interrupts that are activated when there is a level transition on the IRQ line ($0 \rightarrow 1$ or $1 \rightarrow 0$). The last column contains the name of the function that plays the role of the interrupt handler.

"fasteoi" interrupts are level-triggered. Instead of being based on an edge (a signal transition), they depend upon the level of the signal in the interrupt request line. "eoi" stands for "End of Interrupt". The line remains asserted until the interrupt is acknowledged by a CPU. For example, if the interrupt sets the voltage on the line from low to high, then the acknowledgement sets it from high back to low.

IRQ#	CPU 0	CPU 1	CPU 2	CPU 3	HW IRQ type	Handler
0:	7	0	0	0	2-edge	timer
1:	0	0	0	0	1-edge	i8042
8:	0	0	0	0	8-edge	rtc0
9:	0	4	0	0	9-fasteoi	acpi
12:	0	0	0	0	12-edge	i8042
16:	0	0	252	0	16-fasteoi	ehci_hcd:usb1
23:	0	0	0	33	23-fasteoi	ehci_hci:usb2

Table 4.3: Example of a `/proc/interrupts` file

Now, for every request that comes from an IRQ, an interrupt vector is generated. Table 4.4 shows the range of interrupt vectors. NMIs (non-maskable interrupts and exceptions) fall in the range 0-19. The interrupt numbers 20-31 are reserved by Intel for future use. The range 32-127 corresponds to interrupts generated by external sources (typically I/O devices). We are all familiar with interrupt number 128 (0x80 in hex), which is the traditional way to invoke system calls – it is a software-generated interrupt. Most modern machines have stopped using this mechanism because they now have a faster method based on the *syscall* instruction.

239 is the local APIC (LAPIC) timer interrupt. Many IRQs can generate this interrupt vector because there are many timers in modern systems with different resolutions. Finally, the range 251-253 corresponds to inter-processor interrupts (IPIs). A disclaimer is due here. This is the interrupt vector range on the author's Intel i7-based system as of 2023. In all likelihood, this may

change in the future or even be different for other systems. Hence, a request to the reader is to treat this data as just an example.

Interrupt Vector Range	Meaning
0-19	Non-maskable interrupts and exceptions
20-31	Reserved by Intel
32-127	External interrupts
128	System calls
239	Local APIC timer interrupt
251-253	IPIs

Table 4.4: Meaning of interrupt vector ranges

Table 4.5 summarizes our discussion. It shows the IRQ number, interrupt vector and the hardware device for a subset of interrupts. We see that IRQ 0 for the default timer corresponds to interrupt vector 32. The keyboard, system clock, network interface and USB ports have their IRQ numbers and corresponding interrupt vector numbers. One advantage of separating the two concepts – IRQ and interrupt vector – is clear in the case of timers. We can have a wide variety of timers with different resolutions. However, they can be mapped to the same interrupt vector. This will ensure that whenever an interrupt arrives from any one of them, the timer interrupt handler is invoked. The kernel can dynamically decide which timer to use depending on the requirements and load on the system.

IRQ	Interrupt Vector	HW Device
0	32	Timer
1	33	Keyboard
8	40	System clock
10	42	Network interface
11	43	USB port

Table 4.5: IRQ, interrupt vector and HW device

Given that HW IRQs are limited in number, it is possible that we may have more devices than the number of IRQs. In this case, several devices have to share an IRQ. We can do our best to dynamically manage the available IRQs such as deallocating the IRQ when a device is not in use or dynamically allocating an IRQ when a device is accessed for the first time. In spite of all this, we still may not have enough IRQs. Hence, there is a need to share an IRQ between multiple devices. Whenever an interrupt is received from an IRQ, the kernel's interrupt subsystem needs to find which device generated it by running all the handlers corresponding to the connected devices that share the IRQ. These handlers will query the individual devices or inspect the interrupt data and find out which device had raised the interrupt. This inevitably slows down the system, yet is necessary.

Interrupt Handling in Hardware

Let us now go to the next phase, which is the interrupt handler circuitry on a CPU core. It receives the interrupt vector from the LAPIC. This number is between 0-255 and can be used to index the IDT. From the IDT, we get the address of the code segment of the interrupt handler and its base address. After transitioning to kernel mode, the hardware initiates the process of running the interrupt handler. The hardware at this stage is supposed to make a copy of some portions of the running program's context such as the *flags* register and the PC (of the return address). There is some amount of complexity involved. Depending upon the nature of the exception/interrupt, the return address can either be the current program counter or the next one (next PC = either PC + 4 or the branch target). If an I/O interrupt is received, then without doubt we need to store the next PC. However, if there is a page fault, then we need to execute the same instruction once again. In this case, the return address is set to the current PC. It is assumed that the interrupt processing hardware is smart enough to figure all this out. It needs to then store the return address (appropriately computed) and the *flags* register on the user's stack. On x86-64 hardware, the code segment register's contents are also pushed to the stack. This part has to be automatically done in hardware prior to starting the interrupt handler.

4.2.3 Kernel Code for Interrupt Descriptors

struct irq_desc

Listing 4.2: The `struct irq_desc` structure
source : include/linux/irqdesc.h#L55

```
struct irq_desc {
    /* CPU affinity and per-IRQ data */
    struct irq_common_data irq_common_data;

    /* All data w.r.t. the IRQ */
    struct irq_data irq_data;

    /* Pointer to the interrupt flow handler */
    irq_flow_handler_t handle_irq;

    /* Handler, device, flags, IRQ details */
    struct irqaction *action;
}
```

Listing 4.2 shows the important fields in `struct irqdesc`. It is the nodal data structure for all IRQ-related data. It stores all the information regarding the hardware device, the interrupt vector, CPU affinities (which CPUs process it), pointer to an interrupt flow handler, special flags and so on.

irq_common_data stores CPU affinity information and specialized information regarding interrupts that have associated messages (message-signaled interrupts (MSI interrupts)). **irq_data** stores the interrupt vector, IRQ number and other relevant metadata. **handle_irq** is a pointer to a function that is actually a flow handler – it is not a regular interrupt handler. Its job is to collect data

from the device, acknowledge the interrupt, convey information to higher layers (if required) and process the interrupt. The last stage requires the flow handler to first identify the device that had originally raised the interrupt. The devices that are associated with an IRQ are pointed to by the `struct irqaction` structures (refer to Figure 4.6). These structures are organized as a linked list. Let us delve into this further.
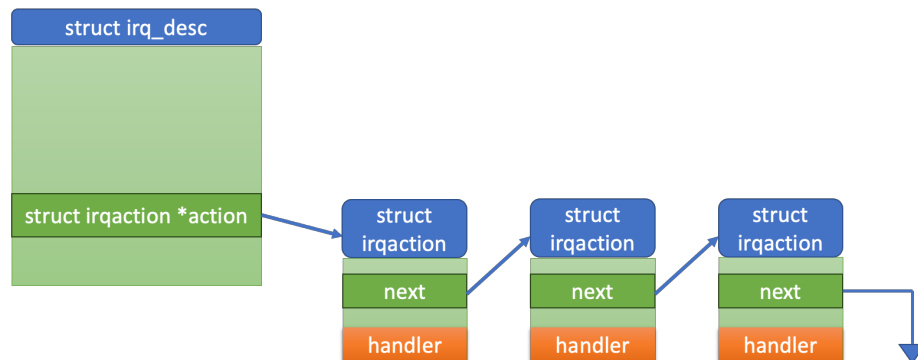
Figure 4.6: The irqaction linked list

struct irqaction

Listing 4.3: `struct irqaction`
source : include/linux/interrupt.h#L118

```
struct irqaction {
    unsigned int        irq;
    irq_handler_t       handler;

    /* associated device */
    void                *dev_id;

    /* arrange as a linked list */
    struct irqaction    *next;

    /* spin off an additional thread */
    irq_handler_t       thread_fn;
    struct task_struct  *thread;
}
```

Listing 4.3 shows the structure of `struct irqaction` and its major fields. It is associated with an irq (`irq`) and points to an interrupt handler, which is typically implemented by the corresponding device driver. It is of type `irq_handler_t` – a function pointer. Additionally, `irqaction` points to a device (`dev_id`). Such `irqaction` structures are organized as a linked list (note the linked list member).

Sometimes, there is a necessity to start a new thread to do additional interrupt processing work. This thread can run with a lower priority and finish the

work at a later point in time. Hence, there is optionally a pointer to a thread (`task_struct`) and a function pointer. The function pointer points to a function that needs to be executed by the thread to perform some *deferred work*.

4.2.4   IRQ Domains

Akin to process namespaces, IRQs are organized in domains. This is especially necessary given that modern processors have a lot of devices, IRQ lines and interrupt controllers. Hence, a hierarchical structure of IRQ domains is needed. Even though we can have a plethora of IRQs, at the end of the day, the processor will only use the interrupt vector (a simple number between 0-255) to access the IDT. It has continued to retain its identity over the years. We will understand the flow of information from an IRQ to the IDT in this section.

A solution similar to hierarchical namespaces is used to manage IRQ domains and IRQs. Within a domain, the IRQ numbers are unique. Recall that we followed a similar logic in process namespaces – within a namespace *pid* numbers are unique. The IRQ number (like a *pid*) is in a certain sense getting *virtualized*. Similar to a namespace's IDR tree whose job was to map *pid* numbers to `struct pid` data structures, we need a similar mapping structure here per domain. It needs to map IRQ numbers to `irq_desc` data structures. Such a mapping mechanism allows us to quickly retrieve an `irq_desc` data structure given an IRQ number. Each IRQ domain is also associated with one or more interrupt controllers. A subset of their pins are mapped to the IRQs in the domain. The function `irq_domain_add` is used to register an IRQ domain with the kernel. Subsequently, interrupt controllers can be added to it. This is similar to adding a process to a namespace first before starting any operation on the process.

In the case of an IRQ domain, the kernel uses a more nuanced solution. If there are less than 256 IRQs in the domain, the kernel uses a simple linear list, otherwise it uses a radix tree. This gives us the best of both worlds. When we have a few IRQs, we avoid the overheads of a radix tree and instead prefer the simplicity of a linked list.

The domains are organized hierarchically. The I/O APIC domains are typically at the leaf level. Their parents are known as *interrupt remapping domains*. Their job is to virtualize multiple I/O APICs. Each such domain forwards the interrupt to the controllers in the LAPIC domain that further virtualize the IRQs, map them to interrupt vectors and present them to the cores. The LAPIC domains are closest to the root.

An astute reader will quickly notice the difference between hierarchical namespaces and hierarchical IRQ domains. In the former, the aim is to make a child process a member of the parent namespace such that it can access resources that the parent owns. However, in the case of IRQ domains, interrupts flow from the child to parent. There is some degree of virtualization and remapping at every stage. For example, one of the domains in the middle could send all keyboard interrupts to only one VM (virtual machine) running on the system. This is because the rest of the VMs may not be allowed to accept inputs from the keyboard. Such policies can be enforced with IRQ domains.

4.2.5 IDT and APIC Initialization Process

> **Point 4.2.1**
>
> The IDT maps the interrupt vector to the address of the handler.

The IDT table is initialized by the BIOS. During the process of the kernel booting up, it is sometimes necessary to process user inputs or other important system events like a voltage or thermal emergencies. Hence, the BIOS needs to run a nanokernel to manage such events. The entries in the IDT point to handlers in the BIOS code. In many cases prior to the OS booting up, the bootloader shows up on the screen; it asks the user about the kernel that she would like to boot. For all of this, we need a bare bones IDT that is already set up. However, once the kernel boots, it needs to overwrite the IDT and adds its own entries. For every single device and exception-generating situation, entries need to be made. These will be custom entries and only the chosen kernel can make them because the BIOS would simply not be aware of them – they are kernel-specific. Furthermore, the interrupt handlers will be in the kernel's address space and thus only the kernel will be aware of their locations. In general, interrupt handlers are not kept in a memory region that can be relocated or swapped out. The pages are locked and pinned in physical memory (see Section 3.1.9).

The kernel uses the `idt_table` data structure to store the IDT. Each entry of this table is indexed by the interrupt vector, and it points to the corresponding interrupt handler. It basically contains two pieces of information: the value of the code segment register and an offset within the code segment. This is sufficient to load the interrupt handler. Even though this data structure is set up by the kernel, it is actually looked up in hardware (like page walks on x86 machines). There is a simple mechanism to enable this. There is a special register called the IDTR register. Similar to the CR3 register for the page table, it stores the base address of the IDT. Thus, the processor knows where to find the IDT in physical memory. A dedicated hardware circuit can *walk* this table, and interrupt handlers can also be automatically loaded by a hardware circuit. The OS need not be involved in this process. Its job is to basically set up the table and let the hardware do the rest.

Setting up the IDT at Boot Time

The main entry point into the kernel, akin to the `main` function in a C program, is the `start_kernel` function defined in `init/main.c`. This master function sets up IDT entries quite early in its execution. It makes a call to `early_irq_init` to probe the default PCI devices and initialize an array of `irq_desc` structures. This probing is done only for setting up devices that one would usually expect such as a graphics card or a network card. These devices are essential to the operation of a modern system.

Next, it makes a call to `init_IRQ` to set up the per-CPU interrupt stacks (Section 3.1.5) and the basic IDT. Once this process is done, the LAPICs and I/O APICs can be setup along with all the connected devices. The `apic_bsp_setup` function realizes this task. All the platform specific initialization functions for x86 machines are defined in a structure `.irqs` that contains a list of function

pointers as shown in Listing 4.4. The function `apic_intr_mode_init` specifically initializes the APICs on x86 machines.

Listing 4.4: The function pointers associated with IRQ handling
source : arch/x86/kernel/x86_init.c#L77

```
.irqs = {
    .pre_vector_init    = init_ISA_irqs,
    .intr_init       = native_init_IRQ,
    .intr_mode_select   = apic_intr_mode_select,
    .intr_mode_init    = apic_intr_mode_init,
    .create_pci_msi_domain  = native_create_pci_msi_domain,
}
```

Setting up the LAPICs

Intel defines a bunch of APIC-related MSRs (model-specific registers) in its architecture. These are privileged registers that are used to interact with interrupt controllers. They are accessible using the `wrmsr` and `rdmsr` instructions. Let us define a few of the important ones.

Logical Destination Register (LDR) Large multi-socket manycore Intel processors can be organized in a 2-level hierarchy. This 32-bit register stores a 16-bit cluster id and a 16-bit processor id (only valid within its cluster). This provides a unique method of addressing a core (especially its LAPIC).

Destination Format Register (DFR) It indicates whether we are following clustering or not.

TPR Task priority register. This stores the priority of the task. When we are dynamically assigning interrupts to cores, the priority stored in this register comes handy.

Initializing a LAPIC in a core includes initializing its full state (all APIC-related MSRs), setting its timers to 0 and finally activating it to receive and process interrupts.

Setting up the I/O APIC

Setting up an I/O APIC is somewhat different [1]. For every single pin in the I/O APIC that is connected to a hardware device, we need to probe the attached devices and set up IRQ data structures for them. Next, for each I/O APIC in the system, there is a need to either create an IRQ domain for it or register it in an existing domain.

4.2.6   The Interrupt Path

Once all the data structures are set up, we are ready to process the interrupt. After saving the context in the default interrupt entry point, the interrupt code pushes the received interrupt vector to the interrupt stack and jumps to the entry point of the IDT.

---

[1]Refer to the code in   arch/x86/kernel/apic/io_apic.c

Listing 4.5: The interrupt entry point
source : arch/x86/kernel/irq.c#L240

```
DEFINE_IDTENTRY_IRQ ( common_interrupt )
{
    ...
    struct irq_desc *desc;
                desc = __this_cpu_read ( vector_irq [ vector ]);
    if ( likely (! IS_ERR_OR_NULL ( desc ))) {
        handle_irq ( desc , regs );
    } else {
     ...
}
```

The code for accessing an IDT entry is shown in Listing 4.5. The `vector_irq` array is the IDT table that uses the interrupt vector (`vector`) as an index to fetch the corresponding `irq_desc` data structure. This array is stored in the per-CPU region, hence the `__this_cpu_read` macro is used to access it. Once we fetch the `irq_desc` data structure, we can process the interrupt by calling the `handle_irq` function. The array `regs` contains the values of all the CPU registers. This was populated in the process of saving the context of the running process that was interrupted. Let us now look at an interrupt handler, referred to as an IRQ handler in the parlance of the Linux kernel. The specific interrupt handlers are called from the `handle_irq` function.

Structure of an IRQ Handler

As we have discussed earlier, there are primarily two kinds of interrupts: level-sensitive and edge-sensitive. They have their separate generic handler functions. For example, the function `handle_level_irq` handles level-sensitive interrupts. It is invoked by `handle_irq`. After a series of calls, all these interrupt handlers ultimately end up invoking the function `__handle_irq_event_percpu`. It is a generic interrupt handler, whose return values are quite interesting. They are as follows.

- NONE: This means that the interrupt was not handled.

- HANDLED: The interrupt was successfully handled.

- WAKE_THREAD: A separate low-priority interrupt handling thread was started. Such threads complete the unfinished work of interrupt handling at a later point in time.

Listing 4.6: Low-level IRQ event handler
source : kernel/irq/handle.c#L139

```
irqreturn_t __handle_irq_event_percpu ( struct irq_desc *desc
    )
{
    irqreturn_t retval = IRQ_NONE;

    /* retrieve the irq number */
    unsigned int irq = desc -> irq_data.irq;
```

```
    struct irqaction *action;

    /* iterate through all the irqactions */
    for_each_action_of_desc (desc, action) {
        /* handle the interrupt */
        irqreturn_t res;
        res = action->handler(irq, action->dev_id);

        /* wake a thread if there is a need */
        switch (res) {
        case IRQ_WAKE_THREAD:
            /* start a separate thread*/
            __irq_wake_thread(desc, action);
            break;

        default: break;
        }
        retval |= res;
    }
    return retval;
}
```

Listing 4.6 shows the code for handling IRQs. Recall that an IRQ can be shared across many devices, which is why we have a linked of `irqaction` structures. The code traverses the entire linked list of `irqaction` structures. Regardless of the return value, the entire linked list is traversed and all the handlers are invoked. In some cases, there is a need to create a separate thread to handle the interrupt. It is possible for a device to register multiple handlers associated with an interrupt vector. All of them are invoked.

All the handlers (of type `irq_handler_t`) are function pointers. They can either point to functions that are generic interrupt handlers defined in the kernel or device-specific handlers defined in the device driver code (the `drivers` directory). Whenever a device is connected in plug-and-play mode or at boot time, the kernel locates the device driver for it. Subsequently, the device driver registers a list of functions with the kernel. One of them is the interrupt handler. It is wrapped in an `irqaction` structure and added to the relevant linked list associated with the interrupt vector.

Top and Bottom Halves

The interrupt handler that does the basic interrupt processing is conventionally known as the *top half*. Its primary job is to acknowledge the receipt of the interrupt to the APIC and urgently service the device's request. Note that such interrupt handlers need to operate in an environment with a lot of restrictions. For some reason, if they are blocked or run for a long time, then they can stall the entire system owing to their high priority.

Interrupt Context

Top-half interrupt handlers run in a specialized *interrupt context*. In the interrupt context, blocking calls such as lock acquisition are not allowed, preemption is disabled, there are limitations on the stack size (similar to other kernel threads) and access to user-space memory is not allowed. They also cannot raise

other interrupts (mostly), cannot perform large memory allocations, print data and perform process-related functions. These are clearly necessary attributes of very high-priority threads that should run and finish quickly.

If the interrupt processing work is very limited, then the basic top-half interrupt handler is good enough. Otherwise, it needs to schedule a bottom-half thread for deferred interrupt processing. A bottom-half thread typically has fewer restrictions. Some variants of bottom-half threads can acquire locks, perform complex synchronization and can take a long time to complete. Moreover, interrupts are enabled when a bottom-half thread is running. This is because such threads have a low priority and there is no risk of the system getting destabilized if they run for a long time.

4.2.7 Exceptions

The Intel processor on your author's machine defines 24 types of exceptions. These are treated exactly the same way as interrupts and similar to an interrupt vector, an exception number is generated.

Even though interrupts and exceptions are conceptually different, they are still handled by the same mechanism, i.e., the IDT. Hence, from the standpoint of interrupt handling, they are the same (they index the IDT in the same manner), however, later on within the kernel their processing paths diverge. Table 4.6 shows a list of some of the most common exceptions supported by the x86 subsystem in the latest version of the Linux kernel.

Trap/Exception	Number	Description
X86_TRAP_DE	0	Divide by zero
X86_TRAP_DB	1	Debug
X86_TRAP_NMI	2	Non-maskable interrupt
X86_TRAP_BP	3	Breakpoint
X86_TRAP_OF	4	Overflow
X86_TRAP_BR	5	Bound range exceeded
X86_TRAP_UD	6	Invalid opcode
X86_TRAP_NM	7	Device not available
X86_TRAP_DF	8	Double fault

Table 4.6: An excerpt from the list of exceptions.
source : arch/x86/include/asm/trapnr.h

Many of the exceptions are self-explanatory. However, some need some additional explanation as well as justification. Let us consider the "Breakpoint" exception. This is pretty much a user-added exception. While debugging a program using a debugger such as *gdb*, we normally want the execution to stop at a given line of code. This point is known as a *breakpoint*. This sadly requires hardware support. Pure software solutions slow down the program significantly. First, it is necessary to include detailed symbol and statement-level information while compiling the binary (known as debugging information). This is achieved by adding the '-g' flag to the *gcc* compilation process. This debugging information maps each line of code to its corresponding program counter value and each variable to its respective memory address. This information is typically stored

in the DWARF format. The debugger subsequently extracts this information and stores it in its internal hash tables.

When the programmer requests the debugger to set a breakpoint corresponding to a given line of code, the debugger finds the program counter that is associated with that line of code and informs the hardware that it needs to stop when it is encountered. Every x86 processor has dedicated debug registers (DR0 ... DR3 plus a few more), where this information can be stored a priori. The processor uses this information to stop at a breakpoint. At this point, it raises the Breakpoint exception, which the OS catches and subsequently lets the debugger know about it. Note that after the processor raises the Breakpoint exception, the program that was being debugged remains effectively paused. The debugger can analyze the state of the running program such as its registers and memory contents. Given that the program is compiled in such a way that at all points of time the mapping between variables and registers/memory addresses is known, it is possible to find the values of all the local and global variables. The user can thus easily find the cause of the bug.

The other exceptions correspond to erroneous conditions that should normally not arise such as accessing an invalid opcode, device or address. An important exception is the "Double fault". It is an exception that arises while processing another exception: it is basically an exception in an exception handler. This indicates a bug in the kernel code, which is never supposed to be there.

Creation of an Entry in the IDT

Let us now look at exception handling (also known as *trap* handling). For every exception, we create an entry in the IDT using the `DECLARE_IDT_ENTRY` macro as shown in Listing 4.7.

Listing 4.7: Declaration of a trap handler
source : arch/x86/include/asm/idtentry.h#L548

```
DECLARE_IDTENTRY(X86_TRAP_DE, exc_divide_error);
```

Here, we are declaring a macro for division-related errors. It is named `exc_divide_error`, and is defined in Listing 4.8. It is important to note that creating an IDT entry is a two-stage process: there is a declaration and a definition. This is a standard design pattern that users must have seen in C programs as well.

Now, in Listing 4.8, we observe that the generic function `do_error_trap` is being invoked. It does the preliminary processing of all exceptions (also known as traps). Along with details of the trap, it takes all the CPU registers (`regs`) as input. Let us quickly explain its arguments. The `regs` argument is a pointer to the stored register state. The next argument is the error code, which in this case is unused. Next is the textual name of the exception, "divide error". `X86_TRAP_DE` is the exception or trap number. It corresponds to the division exception, which can be thrown if the divisor is zero or there is an overflow. The next argument `SIGFPE` is the POSIX error code for a general floating-point exception. `FPE_INTDIV` is an additional argument that indicates this exception arose because of integer division. POSIX is an international standard for an operating system interface. Linux is generally POSIX compliant (not completely

though). Hence, a need was felt to also include the POSIX trap codes in a Linux exception handler. The last argument returns the address of the faulting instruction. In the case of a page fault, we need the virtual address responsible for the fault also. This is automatically stored by the hardware in the `CR2` register. It is an MSR (model-specific register).

Listing 4.8: Definition of a trap handler
source : arch/x86/kernel/traps.c#L205

```
DEFINE_IDTENTRY ( exc_divide_error )
{
     do_error_trap (regs , 0, "divide error", X86_TRAP_DE ,
         SIGFPE , FPE_INTDIV , error_get_trap_addr (regs ));
}
```

Exception Handling

There are several things that an exception handler can do. The various options are shown in Figure 4.7.

Pass it to the User Process
The first option is clearly the most innocuous, which is to simply send a signal to the process and not take any other kernel-level action. Debugging is a prominent example. Here, the processor generates an exception upon the detection of a debug event such as a breakpoint. The kernel is informed. It subsequently sends a signal to the debugging process. Watchpoints (pause when a given address is accessed) also function similarly.

Add a Message to the Logs
The second option is not an exclusive option – it can be clubbed with the other options. The exception handler can additionally print messages to the kernel logs using the built-in `printk` function. This is a kernel-specific print function that writes to the kernel logs. These logs are visible using either the `dmesg` command or are typically found in the `/var/log/messages` file. Many times understanding the reasons behind an exception is very important, particularly when kernel code is being debugged.

Kernel Panic
 The third option is meant for genuinely exceptional cases. Consider a double fault – an exception within an exception handler. This is never supposed to happen unless there is a serious bug in the kernel code. In this case, the recommended course of action is to halt the system and restart the kernel This event is also known as a *kernel panic* (srckernel/panic.c).

Dynamic Binary Translation
 The fourth option is very useful. For example, assume that a program has been compiled for a later version of a processor that provides a certain instruction that an earlier version does not. For example, processor version 10 in the processor family provides the *cosine* instruction, which version 9 does not. In this case, it is possible to create a very easy patch in software such that code that uses this instruction can still seamlessly run on a version 9 processor.

The idea is as follows. We allow the original code to run unmodified. When the CPU will encounter an unknown instruction (in this case the cosine instruc-
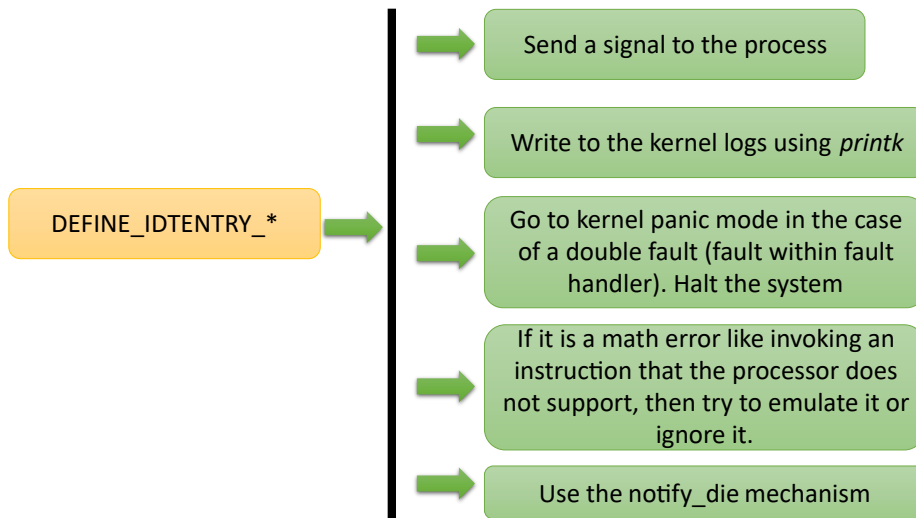
Figure 4.7: Exception handling in Linux
source : arch/x86/kernel/traps.c

tion), it will generate an exception (illegal instruction). The kernel's exception handler can then analyze the nature of the exception and figure out that it is actually the cosine instruction, which is not supported. Killing the entire program for just a single unsupported instruction appears too harsh. In this case, it is possible to use other existing instructions and perform a numerical computation to compute the cosine of the argument and populate the destination register with the result (basically solve the Maclaurin series). The running program can be restarted at the next instruction. The destination register of the cosine instruction will have the correct result. The program will not even perceive the fact that it was running on a CPU that did not support the cosine instruction. Hence, from the point of view of correctness, there is no issue.

Of course, there is a performance penalty – this is a much slower solution as compared to having a dedicated instruction implemented in hardware. However, the code now becomes completely portable across machines. Had we not implemented this dynamic binary translation (or patching) mechanism via exceptions, the entire program would have had to be terminated. A small performance penalty is a very small price to pay in this case.

The notify_die Mechanism

The last option is known as the *notify_die* mechanism, which implements the classic observer pattern in software engineering.

An event like an exception can have multiple processes interested in it. All of them can register and can ask to be notified in case such an exception is raised in the future. All that they need to do is add a callback function (pointer to the exception handler) in a linked list associated with the exception. The callback function (handler) will then be invoked along with some arguments that the exception will produce. This basically means that we would like to associate multiple handlers with an exception. The aim is to invoke them in a certain

sequence and allow all of them to process the exception as per their own internal logic.

Each of these processes that *register* their interest are known as observers or listeners. For example, if there is an error within the core (known as a Machine Check Exception), different handlers can be invoked. One of them can look at the nature of the exception and try to deal with it by running a piece of code to fix any errors that may have occurred. Another interested listener can just log the event. These two exception handlers are clearly doing different things, which was the original intention. We can add more handlers to the chain of listeners, and do many more things. The *notify_die* mechanism simply calls these handlers in sequence.

The return values of the different handlers are quite relevant and important here. This process is similar in character to the `irqaction` mechanism, where we sequentially invoke all the interrupt handlers that share an IRQ line. The return value indicates whether the interrupt was successfully handled or not. In the case of IRQs, we would like to handle an interrupt only once. However, in the case of an exception, multiple handlers can be invoked, and they can perform different kinds of processing. Exception handlers do not enjoy a similar sense of exclusivity. Let us elaborate on this point by looking at the return values of exception handlers that are invoked using the *notify_die* mechanism (shown in Table 4.7). We can either continue traversing the chain of listeners/observers after processing an event or stop calling any more functions. All the options have been provided.

Value	Meaning
NOTIFY_DONE	Do not care about this event. However, other functions in the chain can be invoked.
NOTIFY_OK	Event successfully handled. Other functions in the chain can be invoked.
NOTIFY_STOP	Do not call any more functions.
NOTIFY_BAD	Something went wrong. Stop calling any more functions.

Table 4.7: Status values returned by exception handlers that have subscribed to the *notify_die* mechanism. source : `include/linux/notifier.h`

NOTIFY_DONE indicates that the exception is not relevant for the exception handler. Hence, the next handler in the chain of handlers may be invoked. On the other hand, NOTIFY_OK means that the event was successfully handled. However, the handler was not invoked exclusively. Subsequent handlers in the chain need to be invoked. This means that the process of exception handling hasn't terminated. It terminates when either NOTIFY_STOP is returned or something goes wrong (NOTIFY_BAD).

4.3   Softirqs, Threaded IRQs and Work Queues

Let us now look at a set of bottom-half mechanisms that are used to store and subsequently execute deferred work items. These are used by the top half and bottom half interrupt handlers.

Modern versions of Linux use softirqs and threaded IRQs, which are specialized bottom-half mechanisms to store and execute deferred work items. These are not meant to be used for doing other kinds of regular work (this is their spirit). Linux has a more generic mechanism known as *work queues*. Work queues can be used to execute any generic function as a deferred function call. They run as regular kernel threads in the kernel space. Work queues were conceived to be generic mechanisms.

A brief explanation of the terminology is necessary here. We shall refer to an IRQ using capital letters. A softirq is however a Linux mechanism and thus will be referred to with small letters or with a specialized font `softirq` (when representing a variable in the code).

It is also important to note that the kernel has several execution contexts (or execution modes). The first is the mode in which user processes execute, which is in Ring 3. This is known as the "user context". However, within the kernel itself, we can have different contexts. All regular kernel threads execute in the process context. This includes those kernel threads that are associated with a user process and pure kernel threads that are not associated with any. Top-half handlers run in the interrupt context. Here, all maskable interrupts are disabled and such threads cannot be preempted by other kernel threads. They execute till completion unless they are preempted by higher-priority top-half interrupt handlers. We shall shortly introduce a softirq context that is quite similar to the interrupt context. However, the priority of a softirq handler is effectively lower. Interrupts are enabled, which means that low-priority top-half interrupt handlers can preempt it. However, the softirq handler cannot be preempted by other kernel threads including other threads running in the softirq context.

Definition 4.3.1 Execution Contexts

User Context Typical user-mode process operation (in Ring 3).

Process Context The mode of operation of regular kernel threads that have a priority attached to them. They can either be created out of user processes (temporarily promoted to kernel threads) or can be kernel threads without any user process association. They are the lowest-priority kernel tasks.

Interrupt Context Interrupt top halves run in this context. Maskable interrupts and preemption are disabled. There are a lot of restrictions on the code executing in this context. They cannot access user space memory, cannot make blocking calls, etc. Such threads cannot be swapped out and run until completion. The only exception is if they are interrupted by higher-priority interrupts. In any case, no regular kernel thread in process context gets to run until all the threads in the interrupt context finish.

Softirq Context This is related to the interrupt context. However, in this case interrupts are enabled. Hence, even low-priority interrupts can get serviced. However, no other thread running in the softirq context or any regular kernel thread can swap out the current thread running in softirq context.

4.3.1 Softirqs

A regular interrupt's top-half handler is bound by a large number of rules and constraints regarding what it can and cannot do. A *softirq* inherits the same restrictions and has a lower effective priority because typically interrupts are enabled when it executes (in softirq context). There are two ways that it can be invoked (refer to Figure 4.8).
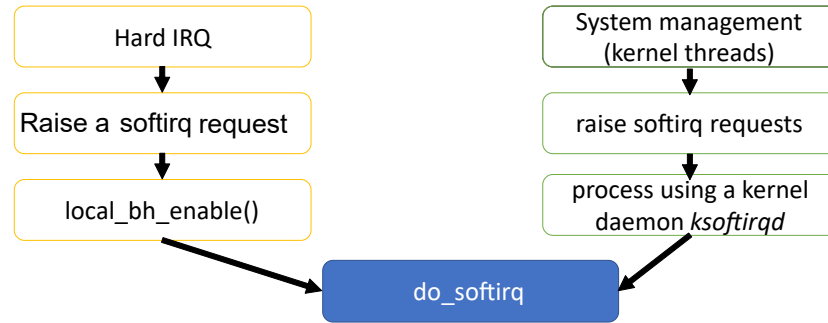

Figure 4.8: Two ways of invoking softirqs

The first method (on the left) starts with a regular I/O interrupt (hard IRQ). After basic interrupt processing, a softirq request is raised. This means that a work parcel is created that needs to be executed either immediately or perhaps later. It is necessary to call the function `local_bh_enable` after this such that the processing of softirq threads can be started. Here "bh" stands for "bottom half".

Subsequently, the function `do_softirq` is invoked. Its job is to check all the deferred softirq work items and execute them one after the other. The thread still runs in interrupt context. However, interrupts are enabled. It is possible that there are a lot of softirq items. Given that no other kernel task can execute, the CPU resources may get monopolized. Hence, there is a need to create a timeout mechanism. This works as follows. Any softirq task in the softirq context runs until completion. It cannot be preempted in middle. However, during its execution, it can raise other softirq requests. Those requests need not be executed after a timeout. Instead, their processing can be deferred till a later point in time.

There is another mechanism for doing this type of work (the right path in the figure). It is not always necessary for top-half interrupt handlers to raise softirq requests. They can be raised by regular kernel threads that want to defer some specialized work for later processing. It is important to note that there may be more urgent needs in the system and thus some kernel work needs to be done immediately. Hence, a deferred work item can be created and stored as a softirq request.

A dedicated kernel thread called `ksoftirqd` runs periodically and checks for pending softirq requests. These threads are called *daemons*. Daemons are dedicated kernel threads that typically run periodically and check/process pending requests. Now, it is interesting to note that `ksoftirqd` runs in process context. It does not run in softirq context. This means that when it calls the function `do_softirq`, it does so in the process context. Any softirq task that it executes

can thus get preempted by other kernel threads. Sadly, it does not enjoy the privileges that it would expect to enjoy in softirq context. `ksoftirqd` also executes those softirq work items, which could not be processed after a top-half handler finished due to a timeout.

The net summary is that softirqs are generic mechanisms that can be used by both top-half interrupt handlers and specialized kernel threads; both can create softirq requests. Some of them execute in interrupt context if they are invoked right after a top-half handler finishes. The deferred work items executed by `ksoftirqd` run in process context.

Raising a softirq

Many kinds of interrupt handlers can raise softirq requests. They all invoke the `raise_softirq` function whenever they need to add a softirq request. Instead of using a software queue, there is a faster method to record this information. A fast method is to store a word in memory in the per-CPU region. Each bit of this memory word has a bit corresponding to a specific type of softirq. If a bit is set, then it means that a softirq request of the specific type is pending at the corresponding CPU.

Table 4.8 shows a few examples of softirqs[2].

Softirq type	Explanation
HI_SOFTIRQ	High-priority software interrupts
TIMER_SOFTIRQ	Timer-related event processing
NET_TX_SOFTIRQ	Network packet transmission
BLOCK_SOFTIRQ	Handle block-device operations
SCHED_SOFTIRQ	Runs the scheduler
HRTIMER_SOFTIRQ	Runs time-sensitive kernel tasks

Table 4.8: Examples of softirqs

As the names suggest, for different kinds of interrupts, we have different kinds of softirqs defined. Note that the size of this list is limited and so is the overall flexibility. The softirq mechanism was never meant to be a generic mechanism in the first place. It was always meant to offload deferred work for a few well-defined classes of interrupts and kernel tasks. It is also not meant to be used by device drivers even though they theoretically can raise softirq requests. We shall see later that work queues are more appropriate for device drivers.

Invoking a softirq Handler

Invoking a handler can either be done after some kernel task finishes like processing the top half of an interrupt or periodically by the kernel daemon (`ksoftirqd`). The processing is quite similar.

It starts with checking all the softirq bits that are set to 1 in the CPU-specific memory word. This means that for each bit set to 1, there is a pending softirq request. Then in a known priority order, the kernel invokes the softirq handlers corresponding to all the bits that are set to 1.

---

[2]Defined in `include/linux/interrupt.h`

4.3.2 Threaded IRQs

Note that softirq threads are still quite restrictive. They are not meant to run for long durations and cannot acquire locks. A mechanism is thus needed to defer work to threads that run as regular processes and do not suffer from any restrictions. This is where threaded IRQs come handy. They have replaced an older mechanism called *tasklets* in terms of use.

They run functions to process deferred work items, albeit using separate kernel threads. The kernel threads that run them still have reasonably high real-time priorities, but these priorities are not as high as interrupt-processing threads that run top-half or softirq tasks. On most Linux distributions, their real-time priority is set to 50, which is clearly way more than all user-level threads and a lot of low-priority kernel threads as well.

We can appreciate this much better by looking at `struct irqaction` again. Refer to Listing 4.9.

Listing 4.9: Relevant part of `struct irqaction`
source : include/linux/interrupt.h#L118

```
struct irqaction {
    ...
    struct task_struct    *thread;
    irq_handler_t         thread_fn;
    ...
}
```

Every `struct irqaction` structure has a pointer to a thread that executes the handler as a threaded IRQ if there is a need. This execution happens in *process context*, where the thread executes as a regular process. It can perform all regular operations like going to sleep, waiting on a lock and accessing user space memory. If this field is NULL, then the IRQ is not meant to be run as a threaded IRQ. Instead, a dedicated interrupt handling thread will process the interrupt.

The next argument of type `irq_handler_t` is a pointer to a function that needs to be executed to handle the interrupt. If the thread argument is not NULL, then a kernel thread executes this function.

4.3.3 Work Queues

Softirqs and threaded IRQs are not very generic mechanisms. Hence, the kernel has work queues that are meant to execute all kinds of deferred tasks. These are generic, flexible and low-priority threads. Work queues have explicitly been designed for this purpose. Their structure is far more elaborate and complex as compared to the rest of the mechanisms.

Broad Overview

Let us provide a brief overview of how a work queue works (refer to Figure 4.9).

A work queue is typically associated with a specific class of tasks such as high-priority tasks, batch jobs or bottom halves. This is not a strict requirement, however, in terms of software engineering, this is a sensible decision.
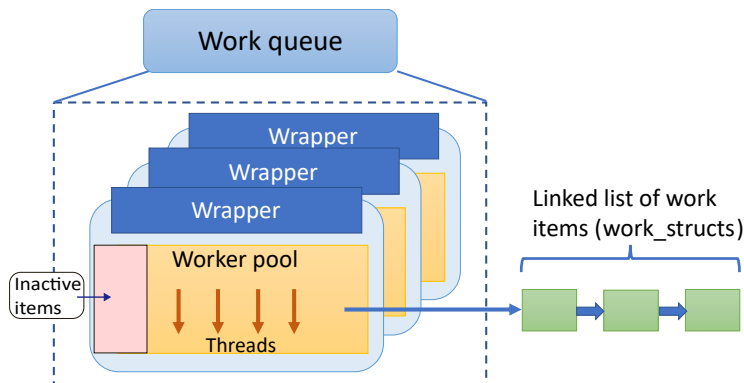
Figure 4.9: Overview of a work queue

Each work queue contains a bunch of worker pool wrappers that each wrap a worker pool. Let us first understand what is a worker pool, and then we will discuss the need to wrap it (create additional code to manage it). A *worker pool* has three components: set of inactive work items, a group of threads that process the work in the pool and a linked list of work items that need to be processed (executed).

The main role of the worker pool is to basically store a list of work items that need to be completed at some point of time in the future. Consequently, it has a pool of ready threads to perform this work and to also guarantee some degree of timely completion. This is why it maintains a set of threads that can immediately be given a work item to process. There is no need to create and destroy threads every time a work item is allocated or completed – a pool of threads is maintained. A work item contains a function pointer and the arguments of the function. A thread executes the function with the arguments that are stored in the work item.

It may appear that all that we need for creating such a worker pool is a bunch of threads and a linked list of work items. However, there is a little bit of additional complexity here. It is possible that a given worker pool may be overwhelmed with work. For instance, we typically associate a worker pool with a CPU or a group of CPUs. It is possible that a lot of work is being added to it and thus the linked list of work items ends up becoming very long. Hence, there is a need to limit the size of the work that is assigned to a worker pool. We do not want to traverse long linked lists.

An ingenious solution to limit the size of the linked list is as follows. We tag some work items as active and put them in the linked list of work items and tag the rest of the work items as inactive. The latter are stored in another data structure, which is specialized for storing inactive work items (meant to be processed much later). The advantage that we derive here is that for the regular operation of the worker pool, we deal with smaller data structures namely the list of active items. It is the role of the wrapper of a worker pool to intercept calls and manage the active and inactive lists.

The worker pool along with its wrapper can be thought of as one cohesive unit. Note that we may need many such *wrapped* worker pools because in a large

system we shall have a lot of CPUs, and we may want to associate a worker pool with each CPU or a group of CPUs. This is an elegant way of partitioning the work and also doing some load-balancing.

Let us now look at the kernel code that is involved in implementing a work queue.

Kernel Code

The important kernel-level data structures and their relationships are shown in Figure 4.10.
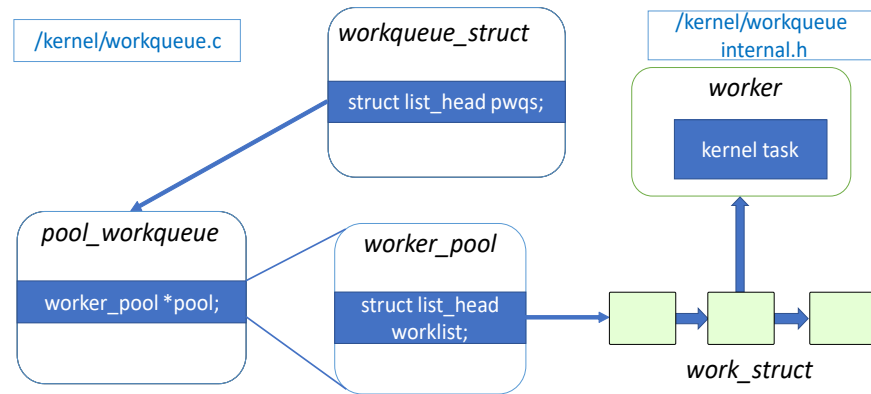

Figure 4.10: The detailed structure of work queues

A work queue is represented using the `workqueue_struct`. It points to a set of *wrapped* worker pools. Note that a work queue needs to have at least one worker pool wrapper (`pool_workqueue`). Each such wrapper points to a `worker_pool`. The wrapper's role is to basically manage the worker pool as discussed earlier. Each worker pool contains a linked list of work items. Each such work item is a parcel of work that is embodied within a structure called `work_struct`.

Each `work_struct` needs to be executed by a worker thread. A worker thread is a kernel task that is a part of the worker pool (of threads). Let us now focus on the fields of `work_struct` (see Listing 4.10).

Listing 4.10: struct `work_struct`
source : include/linux/workqueue.h#L97

```
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
};
```

The member `struct list_head entry` indicates that this is a part of a linked list of work structs. This is per se not a field that indicates the details of the operation that needs to be performed. The only two operational fields of importance are `data` (data to be processed) and the function pointer (`func`). The data field can also be a pointer to an object that contains all the arguments.

The advantage of work queues is that they are usable by third-party code and device drivers as well. This is quite unlike threaded IRQs and softirqs that are not supposed to be used by device drivers. Any entity can create a `struct work_struct` and insert it in a work queue. This is executed later on when there is enough CPU time.

> **Point 4.3.1**
>
> The `workqueue_struct` contains a list of wrappers (`pool_workqueue` structures). Each wrapper wraps a worker pool. Each worker pool is associated with either one CPU or a group of CPUs. It contains a pool of worker threads that process all the constituent work items.

Examples of Some Common Work Queues

Listing 4.11 shows examples of some common work queues defined in the kernel. As we can observe, there are different kinds of work: system-wide work, high-priority tasks, long duration tasks, tasks that have very strict power constraints, tasks that can be inactivated for a long time, etc.

Listing 4.11: Work queues in the system
source : include/linux/workqueue.h#L380

```
extern struct workqueue_struct *system_wq;
extern struct workqueue_struct *system_highpri_wq;
extern struct workqueue_struct *system_long_wq;

/* not bound to a specific CPU */
extern struct workqueue_struct *system_unbound_wq;

/* can be suspended and resumed */
extern struct workqueue_struct *system_freezable_wq;

/* power-efficient jobs */
extern struct workqueue_struct *system_power_efficient_wq;
extern struct workqueue_struct *
   system_freezable_power_efficient_wq;
```

In addition, each CPU has two dedicated work queues: one for low-priority tasks and one for high-priority tasks.

4.4   Signal Handlers

4.4.1   Example of a Signal Handler

Listing 4.12: Code that defines and uses signal handlers

```
1  void handler (int sig){
2     printf ("In the signal handler of process %d \n",
3                              getpid());
4     exit(0);
5  }
```

```
6
7   int main(){
8       pid_t child_pid, wpid; int status;
9
10      signal (SIGUSR1, handler);  /* Register the handler */
11      child_pid = fork();              /*Create the child */
12
13      if (child_pid == 0) {
14          printf ("I am the child and I am stuck \n");
15          while (1) {}
16      } else {
17          sleep (2); /* Wait for the child to get initialized
                   */
18          kill (child_pid, SIGUSR1);  /* Send the signal */
19          wpid = wait (&status);       /* Wait for the child to
                   exit */
20          printf ("Parent exiting, child = %d, wpid = %d,
                   status = %d \n",child_pid, wpid, status);
21      }
22  }
```

Listing 4.12 shows the code of a signal handler. Here, the handler function is `handler` that takes as input a single argument: the number of the signal. Then the function executes like any other function. It can make library calls and also call other functions. In this specific version of the handler, we are making an `exit` call. This kills the thread that is executing the signal handler. However, this is not strictly necessary. These are meant to be generic functions.

Let us assume that we did not make the call to the `exit` library function, but just returned from the handler, then one of the following could have happened: if the signal blocked other signals or interrupts, then their respective handlers would be executed. In the case of some signals such as SIGSEGV and SIGABRT, returning from the handlers is not advisable. They typically indicate a serious issue with the functioning of the program. It is a good idea to save the state, perform a cleanup, release resources and terminate the executing program. For regular signal handlers, the interrupted user thread resumes execution from the point at which it was paused and the signal handler's execution began. From the thread's point of view this is like a regular context switch.

Now let us look at the rest of the code. Refer to the `main` function. We need to register the signal handler. This is done in Line 10. After that, we fork the process. It is important to bear in mind that signal handling information is also copied. In this case, for the child process its signal handler will be the copy of the `handler` function in its address space. The child process prints that it is the child, and then goes into an infinite `while` loop.

The parent process, on the other hand, has more work to do. First, it waits for the child to get fully initialized. There is no point in sending a signal to a process that has not been fully initialized. Otherwise, it will ignore the signal. It thus sleeps for 2 seconds, which is deemed to be enough. It then sends a signal to the child using the `kill` library call that in turn makes the `kill` system call, which is used to send signals to processes. In this case, it sends the SIGUSR1 signal. SIGUSR1 has no particular significance otherwise – it is meant to be defined by user programs for their internal use.

When the parent process sends the signal, the child at that point of time is stuck in an infinite loop. It subsequently wakes up and runs the signal handler. The logic of the signal handler is quite clear – it prints the fact that it is the child along with its process id and then makes the exit call. The parent in turn waits for the child to exit, and then it collects the pid of the child process along with its exit status. The `WEXITSTATUS` macro can be used to parse the exit value (extract its lower 8 bits).

The output of the program shall clearly indicate that the child was stuck in an infinite loop. Then the parent called the signal handler and waited for the child to exit. Finally, the child thread exited.

4.4.2 Signal Delivery

> **Point 4.4.1**
>
> In general, a *signal* is meant to be a message that is sent by the operating system to a process. The signals may be generated by kernel code in response to some hardware interrupt or software event like an exception, or they may be sent by another process (via the kernel). Note that all the signals cannot be blocked, ignored or handled. A signal that cannot be handled like immediate process termination is meant to be exclusively handled by the kernel.

In a multithreaded process that comprises multiple threads, if a signal is sent to it, then one of the threads shall be assigned by the OS to handle the signal. Note that all the signal handling structures are in general shared among all the threads in a thread group. A thread group (also referred to as a multithreaded process) is a single unit insofar as signal handling is concerned. The *kill* command or system call can be used to send a signal to any other process from either the command line or programmatically. Note that *kill* does not mean killing the process as the literal meaning would suggest. It should have been named *send_signal* instead. Let us decide to live with such anomalies ☺. Using the `kill` command on the command line is quite easy. The format is: `kill -signal pid`.

Signal	Number	Description
SIGHUP	1	Sent when the terminal that started the process is closed.
SIGINT	2	The signal generated when we press Ctrl+C. This default action can be overridden in a signal handler.
SIGQUIT	3	Terminates a process. It generates a core dump file (can be used by the debugger to find the state of the process's variables at the time of termination)
SIGILL	4	It is raised when an invalid instruction is executed or the process has inadequate privileges to execute that instruction.

SIGTRAP	5	It is used for debugging. The debugger can program the debug registers to generate this signal when a given condition is satisfied such as a breakpoint or certain other kinds of exceptions.
SIGABRT	6	It is typically generated by library code to indicate an internal error in the program. The signal can be handled but returning to the same point of execution does not make sense because the error in all likelihood will happen again.
SIGBUS	7	It indicates an access to invalid memory. In general, it is raised when there are issues with alignment errors (accessing an integer that starts at an odd-numbered address on some architectures) or other such low-level issues.
SIGFPE	8	It is raised when there is an arithmetic exception such as an overflow or division by zero.
SIGKILL	9	It is a very high-priority signal that causes the program to terminate with immediate effect. It cannot be blocked, ignored or handled.
SIGUSR1	10	This is meant to be used by regular programmers in any way they deem suitable.
SIGSEGV	11	This is similar in character to SIGBUS; however, is far more generic. It is raised when we are trying to dereference a null pointer or accessing memory that is not mapped to a process. It is the most common memory error that C/C++ programmers have to deal with.
SIGUSR2	12	It is similar to SIGUSR1 – it is meant to be used by programmers in their code. This signal is not associated with a fixed set of events.
SIGPIPE	13	This signal is associated with the inter-process communication mechanism where two processes use a *pipe* (similar to a FIFO queue) to communicate between themselves. If one end of the pipe is broken (process terminates or never joins), then this signal is raised by the OS.
SIGALRM	14	A process can set an alarm using any of the timer chips available in the hardware. Once the time elapses, the OS raises a signal to let the process know. It works like a regular alarm clock.
SIGTERM	15	It is a signal that causes process termination. It is a "polite" way of asking the program to terminate. It can be blocked, ignored or handled.
SIGSTKFLT	16	This signal is very rarely used these days. It stands for "stack fault". It is used to indicate memory access problems in the stack segment of a process. SIGSEGV has replaced this signal in modern kernels.

SIGCHLD	17	When a child process terminates, this signal is sent to the parent.
SIGCONT	18	This resumes a stopped process.
SIGSTOP	19	It stops a process. It has a very high priority. It cannot be caught, handled or ignored.
SIGTSTP	20	It is a polite version of SIGSTOP. This signal can be handled. The application can be stopped gracefully after the handler performs some bookkeeping actions.

Table 4.9: List of common signals including their definitions
source : include/uapi/asm − generic/signal.h

Refer to Table 4.9 that shows some of the most common signals used in the Linux operating system. Many of them can be handled and blocked. However, there are signals such as SIGSTOP and SIGKILL that cannot be handled, blocked and ignored. The kernel directly stops or kills the associated processes, respectively.

In modern kernels, there are different ways of sending a signal to a thread group. One of the simplest approaches is the `kill` system call that can send any given signal to a thread group (as we have seen in Listing 4.12). One of the threads handles the signal. There are many versions of this system call. For example, the `tkill` call can send a signal to specific thread within a thread group, whereas the `tgkill` call takes care of a corner case. It is possible that the thread id specified in the `tkill` call is recycled. This means that the thread completes, and then a new thread is spawned with the same *pid*. This can lead to the signal being sent to the wrong thread. To guard against this rare case, the `tgkill` call takes an additional argument, the thread group id. It is unlikely that both will be recycled and still remain the same.

SIGKILL and SIGSTOP are special in other ways as well. Even though signals are generally sent to a specific thread in a thread group, these signals are sent to all the threads. This is because they are meant to affect the entire thread group. They either destroy all the threads or stop all of them (resp.).

Regardless of the method that is used and the nature of the signal, it is very clear that signals are sent to a thread group; they are not meant to be sent to a particular thread unless the `tkill` or `tgkill` calls are used. An example of thread-specific handling is as follows. Sometimes, there is an arithmetic exception in a thread and there is a need to call the specific handler for that thread only. In this case, it is not possible nor advisable to call the handler associated with another thread in the same thread group.

Furthermore, signals can be blocked as well as ignored. When a blocked signal is raised, it is queued. All such queued/pending signals are handled once they are unblocked. Here also there is a caveat: no two pending signals of the same type can be pending for a process at the same time. Moreover, when a signal handler executes, it *blocks* the corresponding signal.

> **Point 4.4.2**
>
> No two pending signals of the same type can be pending for a process at the same time. Moreover, when a signal handler executes, it *blocks* the corresponding signal.

More about Signal Handling

There are several ways in which a signal can be handled.

The first option is to ignore the signal – it means that the signal is not important, and no handler is registered for it. In this case, the signal can be happily ignored. On the other hand, if the signal is important and must lead to process termination, then the process needs to be terminated. Examples of such signals are SIGKILL and SIGINT (refer to Table 4.9). There can also be a case where process termination is inevitable. However, prior to terminating the process, an additional file called the *core dump* file needs to be generated. It contains the entire memory and register state of the process. It can be used by a debugger to inspect the state of the process at which it was paused or stopped because of the receipt of a signal. For instance, we can find the values of all the local variables, the stack's contents and the memory contents.

We have already seen the process stop and resume signals earlier. The stop action is associated with suspending a process indefinitely until the resuming action is initiated. The former corresponds to the SIGSTOP and SIGTSTP signals, whereas the latter corresponds to the SIGCONT signal. It is important to understand that like SIGKILL, these signals are intercepted by the kernel and the corresponding set of threads in the thread group are either all terminated or stopped/resumed. SIGKILL and SIGSTOP in particular cannot be ignored, handled or blocked.

Finally, the last method is to handle the signal by registering a handler. Applications that provide a graphical user interface often use signal handlers to process keyboard and mouse click events. For example, when a mouse button is clicked, the kernel catches the interrupt and raises a signal to let the foreground application know about the mouse click. The relevant signal handler runs and processes the event. It can, for instance, open a new window or make a change to some visual element.

Note that in many cases this may not be possible, especially if the signal arose because of an exception. The same exception-causing instruction will execute after the handler returns and again cause an exception. In such cases, terminating or stopping the faulting thread are good options. In some cases, if the circumstances behind an exception can be changed, then the signal handler can provide an effective solution. For example, it can remap a memory page or change the value of a variable that is causing an exception. Making such changes in a signal handler to fix the state of a running program is quite risky and is only meant for black belt programmers ☺.

4.4.3   Kernel Code

Let us now look at the relevant kernel code (shown in Listing 4.13). It contains the fields in the `task_struct` that pertain to signal handling.

Listing 4.13: Fields in the `task_struct` that pertain to signals
source : include/linux/sched.h#L1098

```
/* signal handling apex structure */
struct signal_struct    *signal;

/* list of all the handlers */
struct sighand_struct   *sighand;

/* currently blocked and originally blocked signals */
sigset_t                blocked;
sigset_t                real_blocked;

/* list of all the pending signals */
struct sigpending       pending;

/* custom signal stack */
unsigned long           sass_ss_sp;
size_t                  sass_ss_size;
```

The apex data structure is `signal_struct`. It holds all the details about the threads involved in signal handling and the list of pending signals. The information about the registered signal handlers is kept in `struct sighand_struct`. The two important fields that store the set of blocked/masked signals are `blocked` and `real_blocked`. They are of the type `sigset_t`, which is nothing but a bit vector: one bit for each signal (that has been raised). It is possible that a lot of signals have been blocked by the process because it is simply not interested in them. All of these signals are stored in the variable `real_blocked`. During the execution of any signal handler, typically more signals are blocked including the signal that is being handled. There is a need to add all of these additional signals to the set `real_blocked`. With these additional signals, the expanded set of signals is called `blocked`.

Hence, we have the following relationship.

$$\text{real\_blocked} \subset \text{blocked} \tag{4.1}$$

In this case, we set the `blocked` signal set, which is a super set of the set `real_blocked`. These are all the signals that we do not want to handle when a signal handler is executing. After finishing executing the handler, the kernel sets `blocked` to `real_blocked`.

`struct sigpending` stores the list of pending/queued signals that have not been handled by the process yet. We will discuss its intricacies later.

Finally, consider the last two fields, which specify the details of an alternative stack. For a signal handler, we may want to use the same stack of the thread that was interrupted or a different one. If we are using the same stack, then there is no problem; we can otherwise use a different stack in the thread's address space. In this case its starting address and the size of the stack need to be specified. If we are using the alternative stack, which is different from the real stack that the thread was using, no correctness problem is created. The original thread in any case is stopped and thus the stack that is used does not matter.

struct signal_struct

Listing 4.14: Fields in `signal_struct`
source : include/linux/sched/signal.h#L93

```
struct signal_struct {
    /* number of active threads in the group */
    atomic_t live;

    /* all the threads in the thread group */
    struct list_head thread_head;

    /* threads waiting on the wait system call */
    wait_queue_head_t  wait_chldexit;

    /* last thread that received a signal */
    struct task_struct *curr_target;

    /* shared list of pending signals in the group */
    struct sigpending shared_pending;
};
```

Listing 4.14 shows the important fields in the main signal-related structure `signal_struct`. It contains process-related information such as the number of active threads in the thread group, a linked list containing all the threads (in the thread group), list of all the constituent threads that are waiting on the `wait` system call, the last thread that processed a signal and the list of pending signals (shared across all the threads in a thread group).

We need to understand that in general the kernel does not attach any special significance to threads. The scheduler and other parts of the kernel, by and large, treat each thread as an independent process. It has its own *pid*. The signal subsystem is a noteworthy exception. Here, a lot of information is maintained regarding the status of all the threads in a thread group, whether they are waiting for a child to terminate or not, and load-balancing information. For example, the id of the last thread that processed a signal is maintained. Next time, another thread could be assigned to process the signal such that all the threads are equally slowed down. Also note that pending signal information is stored at the level of a thread group, not at the level of individual threads. Next, let us discuss the structure that maintains the details of the signal handlers, which are also shared across the threads. Given that all the threads in a thread group share the virtual address space, the same virtual address points to the same function, which in this case is a signal handler.

struct sighand_struct

Listing 4.15: Fields in `sighand_struct`
source : include/linux/sched/signal.h#L20

```
struct sighand_struct {
    refcount_t       count;
    wait_queue_head_t  signalfd_wqh;
    struct k_sigaction  action[_NSIG];
};
```

Listing 4.15 shows the `sighand_struct`, which serves as a wrapper of signal handlers.

The first field `count` maintains the number of `task_struct` s that use this handler. Reference counting of this nature is a common design pattern in the kernel. When the count reaches zero, it means that there are no processes with registered signal handlers. The next field `signalfd_wqh` is a queue of waiting processes. At this stage, it is fundamental to understand that there are two ways of sending a signal to a process. We have already seen the first approach, which involves calling the signal handler directly. This is a straightforward approach and uses the traditional paradigm of using *callback functions*, where a callback function is a function pointer that is registered with the caller. In this case, the caller (invoker) is the signal handling subsystem of the OS.

It turns out that there is a second mechanism, which is not used that widely. As compared to the default mechanism, which is asynchronous (signal handlers can be run any time), this is a synchronous mechanism. In this case, signal handling is a planned process. It is not the case that signals can arrive at any point of time, and then they need to be handled immediately. The idea is that the process registers a file descriptor with the OS – we refer to this as the *signalfd* file. Whenever a signal needs to be sent to the process, the OS writes the details of the signal to the *signalfd* file. Threads in this case, typically wait for signals to come (get queued in `signalfd_wqh`). When a signal arrives, the relevant thread is woken up. If there is already a waiting signal, the invoking thread can pick the details and start processing the signal. The locus of control is transferred to the thread.

The asynchronous mechanism is more commonly used. Threads are immediately notified if there is a signal. This allows them to be responsive, especially with graphical user interfaces. Hence, let us continue our journey in describing the key structures associated with it.

For signal handing, we need to store an array of `_NSIG` (set to 64) signal handlers. 64 is the maximum number of signal handlers that Linux supports on x86 systems. Each signal handler is wrapped using the `k_sigaction` structure. On most architectures, this simply wraps the `sigaction` structure, which we shall describe next.

struct sigaction

Listing 4.16: `struct sigaction` (x86 systems)
source : arch/x86/include/uapi/asm/signal.h#L94

```c
struct sigaction {
    /* pointer to the handler */
    __sighandler_t sa_handler;
    unsigned long sa_flags;

    /* additional signals masked */
    sigset_t sa_mask;
};
```

The important fields of `struct sigaction` are shown in Listing 4.16. The fields are reasonably self-explanatory. `sa_handler` is a function pointer in the thread's user space memory. `flags` represents the parameters that the kernel

uses to handle the signal such as whether a separate stack needs to be used or not. Finally, we have the set of signals that are additionally masked when the handler is running.

struct sigpending

The final data structure that we need to define is the list of pending signals (`struct sigpending`). This data structure is reasonably complicated. It uses some of the tricky features of linked lists, which we have very nicely steered away from up till now.
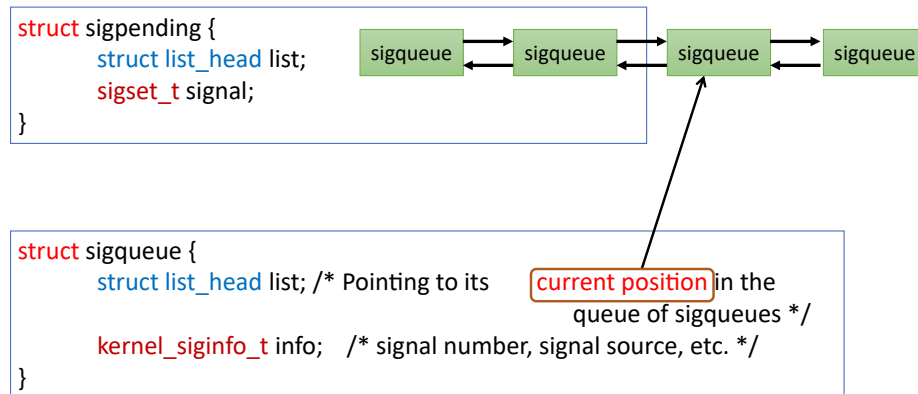


```
struct sigpending {
        struct list_head list;
        sigset_t signal;
}
```

```
struct sigqueue {
        struct list_head list; /* Pointing to its current position in the
                                  queue of sigqueues */
        kernel_siginfo_t info;   /* signal number, signal source, etc. */
}
```

Figure 4.11: Data structures used to store pending signals

Refer to Figure 4.11. The structure `sigpending` wraps a linked list that contains all the pending signals. The name of the list is as simple as it can be, `list`. The other field of interest is `signal` that is simply a bit vector whose i^{th} bit is set if the ith signal is raised. Note that this is why there is a requirement that two signals of the same type can never be pending for the same process. We just have a single bit to record the fact that a signal has been raised. Hence, two signals of the same type cannot be outstanding at the same point of time.

Each entry of the linked list is of type `struct sigqueue`. Note that we discussed in Appendix C that in Linux, different types of nodes can be part of a linked list. Hence, in this case we have the head of the linked list as a structure of type `sigpending`, whereas all the entries are of type `sigqueue`. As non-intuitive as this may seem, this is indeed possible in Linux's linked lists.

Each `sigqueue` structure basically functions as a node of a linked list. Hence, it is mandated to have an element of type `struct list_head`. Recall that a *struct list_head* points to linked list nodes on the left and right (previous and next), respectively. Each such `sigqueue` encapsulates a raised signal using the `kernel_siginfo_t` structure (kernel signal information).

This structure contains the following fields: signal number, number of the error or exceptional condition that led to the signal being raised, source of the signal and the sending process's pid (if relevant). This is all the information that is needed to store the details of the signal that has been raised, and process it later.

> **Trivia 4.4.1**
>
> If n bits are set (equal to 1) in the field `signal`, then it means that there are n signals raised. For each raised signal, we have an entry in the linked list whose head is the `sigpending` structure. Each such entry stores the details of the signal and is represented by a `sigqueue` structure.

4.4.4 Entering and Returning from a Signal Handler

A signal is similar to a context switch. The executing thread is stopped, and the signal handler is run. We are, of course, assuming that we are using the default version of signal handling and not the file-based method. Assuming the default method, the first task is to save the context of the user process.

Kernel routines that are specialized to save the context are used to collect the context of the running process. This part is similar to a system call or interrupt. However, the difference is that the context is not stored on the kernel stack. There is no need to do so given that we don't expect a kernel thread to subsequently run. An elaborate low-level data structure is created to store the context on the user stack, which is often referred to as the signal frame. The data structures to capture the context are shown in Listing 4.17.

Listing 4.17: User thread's context stored by a signal handler
source : arch/x86/um/signal.c#L349
source : include/uapi/asm − generic/ucontext.h#L5

```
struct rt_sigframe {
    struct ucontext uc;        /* context */
    struct siginfo info;       /* kernel_siginfo_t */
    char __user *pretcode;     /* return address:
        __restore_rt glibc function */
};

struct ucontext {
    unsigned long uc_flags;
    stack_t uc_stack;          /* user's stack pointer */
    struct sigcontext uc_mcontext; /* Snapshot of all the
        registers and other state */
};
```

struct `rt_sigframe` keeps all the information required to store the context of the thread that was signaled. The context per se is stored in the structure struct `ucontext`. Along with some signal handling flags, it stores two vital pieces of information: the pointer to the user thread's stack and the snapshot of all the user thread's registers and its state. The stack pointer can be in the same region of memory as the user thread's stack or in a separate memory region. Recall that it is possible to specify a separate address for storing the signal handler's stack.

The next argument `info` is the signal information that contains the details of the signal: its number, the relevant error code and the details of the source of the signal.

The last argument `pretcode` is the most interesting. The question is where should the signal handler return to? It cannot return to the point at which the original thread stopped executing. This is because its context has not been restored yet. Hence, we need to return to a special function that needs to do a host of things such as restoring the user thread's context. Hence, here is the big idea. Before launching the signal handler, we deliberately tweak the return address such that the handler returns to a custom function that can restore the user thread's context. Note that on x86 machines, the return address is stored on the stack prior to invoking any function. All that we need to do is change the corresponding entry on the stack and make it point to a specific function: `__restore_rt` function in the glibc standard library.

When the signal handler returns, it will start executing the `__restore_rt` function. Note that there is no reason to write the signal handling function in any special manner. It is completely oblivious of such changes. Tweaking the return address by modifying the stack or the return address register is a standard technique. This idea find many uses in OS kernels. For example, there is typically a need to execute a function on a separate thread. It returns to a special code snippet that records the return value and tears down the thread.

Now, the `__restore_rt` function does a lot of important things. It does some bookkeeping and makes the important `sigreturn` system call. This transfers control back to the kernel. It is only the kernel that can restore the context of a process. This cannot be done in user space without hardware support. Hence, it is necessary to bring the kernel into the picture. The kernel's system call handler copies the context stored in the user process's stack using the `copy_from_user` function to the kernel's address space. The same way that we restore the context while loading a process on a core, we do exactly the same here. The context collected from user space is transferred to the same subsystem in the kernel; it restores the user thread's context (exactly at where it stopped). The kernel populates all the registers of the user thread including the PC and the stack pointer. Ultimately, the user thread starts from exactly the same point at which it was paused to handle the signal.

To summarize, a signal handler is a small process within a process. It has a short-lived life. It ceases to exist after the signal handling function finishes its execution. Subsequently, the original thread resumes.

4.5 Summary and Further Reading

4.5.1 Summary

Summary 4.5.1

1. A library call wraps a system call. It prepares its arguments, invokes it, processes the return value and informs the user application accordingly.

2. Library calls such as `printf` are implemented in multiple layers arranged sequentially. Each layer processes the output of the previous layer. Ultimately, in the case of `printf`, a single string is created and sent as an argument of the system call.

3. On x86 machines, the system call number is sent via the `rax` register. The rest of the six arguments are sent via other registers. If there are additional arguments, they are sent via the user stack.

4. System calls use the `syscall` instruction to enter the kernel. After a mode switch, the context is first saved and then system call processing begins.

5. Interrupts and exceptions are associated with an 8-bit number known as the interrupt vector. This is used to access the Interrupt Descriptor Table (IDT). Each entry stores a pointer to an `irq_desc` structure, which the kernel uses to locate the handler.

6. Every device is associated with an interrupt line known as an IRQ. These IRQs can be physical copper wires or could be virtual. Every device is connected to an Advanced Programmable Interrupt Controller (APIC). The APIC can raise an IRQ (indicate that it is set to 1) upon either sensing a voltage change on the IRQ line or on receiving a message.

7. The APICs themselves are organized hierarchically (in domains). There are a few system-wide I/O APICs, and every core has a local APIC (LAPIC). The LAPIC sends the interrupt vector to the CPU and also provides other services such as maintaining a timer that can generate periodic interrupts or work as an alarm.

8. There is a limit of 256 interrupt vectors, and a modern motherboard can potentially have many more devices and IRQs. Hence, there is a need to dynamically map IRQs to interrupt vectors, and potentially multiplex an interrupt vector among several IRQs. There is also a need to share an IRQ between several devices.

9. When an interrupt is raised, the kernel can get a pointer to the IRQ via the corresponding entry in the IDT. Subsequently, it is necessary to query every device associated with the IRQ and find if it had raised the interrupt. If the device agrees, then its device driver's handler is invoked.

10. The interrupt handler is known as the *top half*. It is a very high-priority task. Its priority is much more than all normal and real-time tasks. Because of such high priorities, there need to be restrictions on top-half handlers. They cannot acquire locks, cannot access user-space memory, and dynamically allocate memory. Moreover, during their execution, interrupts are disabled. Hence, if there is work of a more generic nature, it needs to be deferred for later processing. Another task needs to pick up the work, which is known as a *bottom-half* handler.

11. When an exception is detected, there are several things that the kernel can do. Note that exceptions are often generated when there is some bug in the program. The following options are not exclusive.

(a) Send a signal to the process and let it handle the exception.

(b) Print to the kernel logs using the `printk` function.

(c) If there is a fault within an exception handler, then this situation is known as a double fault. The kernel in this case goes into *panic* mode and the system shuts down.

(d) Perform dynamic binary translation and replace the faulting instruction with a software-generated code snippet.

(e) Use the *notify_die* mechanism to inform entities that have indicated interest in getting notified about this exception.

12. There are three methods to implement bottom-half handlers.

Softirqs Sometimes the top-half handler finishes and raises a softirq. It can immediately start executing the softirq request in the interrupt context. In this case, interrupts are enabled and the softirq thread has a lower priority than all top-half interrupt handlers.

Threaded IRQs These threads run at a high real-time priority, which is typically 50.

Work Queues This is the most generic mechanism and can be used by all subsystems. In this case, the worker threads run with normal kernel process priorities.

13. The `signal` call is used to register signal handlers and the `kill` call is used to send a signal to a process.

14. At any point of time only one signal of a given type can be raised. The kernel has elaborate data structures to maintain all signal-related information such as the list of waiting threads, pending signals and registered handlers.

15. Every signal handler returns to a custom location and starts executing the `__restore_rt` function. It makes the `sigreturn` system call, which copies the saved context from the user process's stack to kernel space, and subsequently directs the kernel to restore the context of the user process and resume it.

4.5.2 Further Reading

Exercises

Ex. 1 — What is the need to organize APICs as domains?

Ex. 2 — How are arguments passed to system calls? What happens if we have a lot of arguments?

Ex. 3 — What are top-half and bottom-half interrupt handlers in Linux? What are their advantages?

Ex. 4 — The way that we save the context in the case of interrupts and system calls is slightly different. Explain the nature of the difference. Why is this the case?

Ex. 5 — If we want to use the same assembly language routine to store the context after both an interrupt and a system call, what kind of support is required (in SW and/or HW)?

Ex. 6 — Why is the signal context stored on the user stack?

Ex. 7 — Consider a signal handler that is registered by a multithreaded user process. When the signal is delivered to the user process, there are a host of options for processing the signal. Comment on the following options and their relative pros and cons.

   i)Deliver the signal to any one of the threads.
  ii)Create a separate thread for the signal handler.
 iii)Deliver the signal to all the threads.

Ex. 8 — Consider the case of a signal handler – a function that is registered with the operating system that the OS needs to invoke when it needs to send a signal to a process.

   a)The arriving signal causes a new function to run in the address space of a process by interrupting its execution. Should it use the same stack or a different stack? What are the pros and cons?
   b)For the signal handler to take any effect, it needs to make changes to global variables. How should the programmer deal with such asynchronous events?
   c)Can a graphical user interface that takes input from the mouse benefit from signal handlers?
   d)How is a signal handling function expected to complete? Where will it return to and how?

Ex. 9 — Answer the following questions regarding signal handlers.

    a)Do *struct sigpending* and *struct sigqueue* reference the same data structure? Explain.

    b)How is the return address of a signal handler set? What is it set to?

Ex. 10 — What is the philosophy behind having sets like `blocked` and `real-blocked` in signal-handling structures? Explain with examples.

Ex. 11 — How does the interrupt controller ensure real-time task execution on an x86 system? It somehow needs to respect real-time process priorities.

Ex. 12 — What is the need to share IRQs between devices? How do we ultimately find out which device raised an interrupt? Explain the low-level details as well.

Ex. 13 — How is the use of softirqs restricted? What is the need for this restriction?

Ex. 14 — What is the need for having a specific handler such as a softirq and a generic handler such as a workqueue?

Ex. 15 — What are the beneficial features of softirqs and threaded IRQs?

Ex. 16 — Why is the `notify_die` mechanism useful?

Ex. 17 — If an instruction is not supported, there is an illegal instruction exception. However, the exception handler can sometimes <u>fix</u> the problem. Answer the following questions in this context:

    a)When is this facility useful?

    b)How does the exception handler typically perform the "fix"?

    c)Where does it return to?

Chapter 5

Synchronization and Scheduling

In this chapter we will discuss one of the most important concepts in operating systems namely synchronization and scheduling. Synchronization deals with managing resources that are common to a bunch of processes or threads (shared between them). It is possible that there will be competition among the threads or processes to acquire a resource: this is also known as a *race condition*. Such races can lead to errors and undefined behavior. As a result, there is a need to enforce certain restrictions. For example, it is often necessary to allow only one thread to access a shared resource at a time. This is known as *mutual exclusion*. It is one of the simplest examples of synchronization. There may be a need to enforce more complex conditions such as all the threads need to arrive at a certain point before any thread is allowed to proceed (a barrier). Another example could be a simple producer-consumer queue. The producer thread can add items to the queue till it fills up. Subsequently, it needs to wait to block and wait for the consumer thread to dequeue items. Similarly, the consumer thread needs to block if the queue is empty. This is another example of synchronization that requires a complex interaction between threads. There is often a need for such patterns because a lot of tasks on modern systems require multiple processes and unless they effectively interact with each other, complex objectives cannot be realized. The kernel needs to facilitate such interactions.

The kernel is also not a monolith. It has a lot of concurrently running threads that often run in parallel on different cores. As a result, all of its internal data structures have to be *thread safe*. This means that they need to allow concurrent accesses by multiple threads. Furthermore, these concurrent accesses may be write accesses that change the state of the data structure. We shall observe that there are different ways for ensuring that concurrent accesses remain safe. The simplest mechanism is locking, which is a way of allowing only on thread execute a piece of code known as a *critical section*. However, locks are the simplest mechanisms in this space. There are more complex mechanisms that enable the kernel to efficiently discard old versions of a data item, and there are strategies that avoid such locks altogether.

Once all such synchronizing conditions have been designed and implemented, it is the role of the kernel to ensure that all the computing resources namely the cores and accelerators are optimally used. There should be no idleness

or excessive context switching. Therefore, it is important to design a proper scheduling algorithm such that tasks can be efficiently mapped to the available computational resources. We shall see that there are a wide variety of scheduling algorithms, constraints and possible scheduling goals. Given that there is such a diversity of use cases and there are so many practical scenarios possible, there is no one single universal scheduling algorithm that outperforms all the others. In fact, we shall see that for different types of problems, we need to have different types of scheduling algorithms. Some are well suited for resource-constrained environments, some work well in large multicore systems and some perform well in systems that try to minimize context switching.

Subsequently, it is necessary to consider real-time systems that associate deadlines with tasks. In soft real-time systems, deadlines can be occasionally violated. However, in hard real-time systems, it is not possible to violate deadlines. Hard real-time systems are deployed on missiles, controllers in nuclear reactors and healthcare systems. In such systems, violating a deadline can have lethal consequences. It is thus necessary to re-architect the kernel to run such systems. Furthermore, it should be possible to make theoretical guarantees of the following nature: if the load on the system is below a certain threshold, no deadlines will be violated. This is an established area now and the entire field of real-time systems has led to a plethora of real-time kernels including Linux RT (real-time version of the Linux kernel). Along with supporting real-time scheduling and resource management, almost all the parts of the kernel are preemptible. This means that it is possible for higher priority tasks to displace them. They do not allow long executions with interrupts disabled. This would mean that a low-priority interrupt handler can stop a real-time task from executing. This is not permissible. Hence, disabling interrupts should either be avoided altogether or should be done as infrequently as possible. Even if interrupts have to be disabled, the code executed in such regions needs to be as little as possible.

Organization of this Chapter

Figure 5.1 shows the organization of this chapter.

We will start with discussing the basics of synchronization. It is important to realize that there is a need to wrap shared variables in critical sections because of the possibility of data races. Hence, there is a need to use locks such that critical sections can be created. We will subsequently move on to discussing *pthreads* – the most popular threading library on Linux systems. In the context of pthreads, we will discuss semaphores and condition variables, which are advanced synchronization primitives. The advantage of using synchronization primitives is that concurrent data structures can be created such as concurrent producer-consumer queues that are heavily used in the kernel. We thus have a section dedicated to queues.

The next section deals with kernel-level concurrency. Spinlocks and kernel mutexes are two of the most basic mechanisms in the kernel. They are used to implement complex data structures and synchronization mechanisms. It is possible that in this process deadlocks can get introduced, where a set of processes wait on each other to release their locks. Hence, there is a need for a dedicated mechanism in the kernel to detect and recover from deadlocks. This is known as the *lockdep* mechanism. Another important issue that arises in concurrent
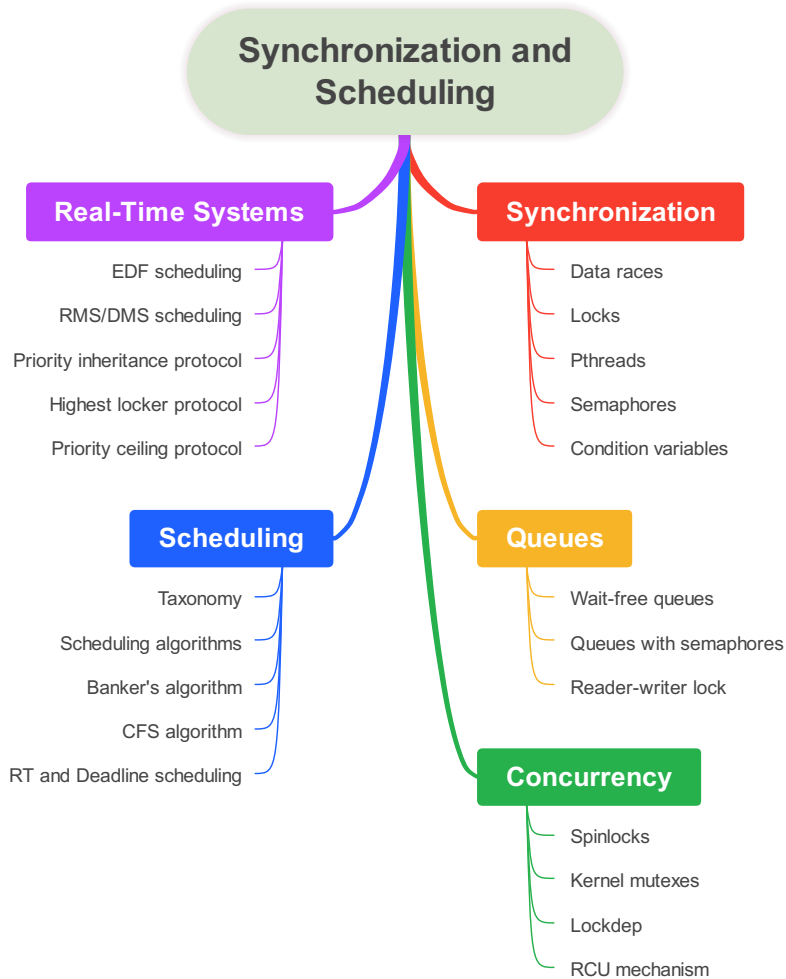
Figure 5.1: Organization of the chapter

kernels is that when a pointer is updated to point to a new object, the object that was previously pointed to needs to be garbage collected. However, prior to doing so we need to ensure that no process holds active references to it. Hence, this process needs to be automated such that it is easy to switch pointers and perform automatic garbage collection to some extent. The Linux kernel is famous for implementing the read-copy-update (RCU) mechanism, which we shall cover in detail. It solves this problem.

Next, we will move on to discuss scheduling. Instead of looking at simple variants of the problem that have no practical significance, we shall directly proceed to discussing a taxonomy of all scheduling problems. We shall observe that

there is a plethora of problems. Many of them have fairly simple optimal solutions, even though proving optimality may not be that simple. However, a large number of problems in this space are provably NP-complete. We shall discuss a few of the common heuristics in this space, especially algorithms that are provably deadlock-free. The Banker's algorithm is quite popular in this space. It considers multiple identical copies of resources and mimics real-world scenarios closely. Next, we shall discuss the Linux kernel's native CFS (Completely Fair Scheduling) algorithm. It balances priority with fairness. Finally, we shall move on to discussing Linux's real-time scheduling classes: priority-based scheduling and deadline scheduling.

In the last section, we shall discuss real-time systems and associated scheduling algorithms. In the world of real-time systems the aim is to design scheduling algorithms where every task is guaranteed to finish by its deadline, subject to the system load being below a threshold. Different guarantees can be made based on whether tasks are preemptible or not. We shall discuss the EDF (Earliest Deadline First) and the Rate/Deadline Monotonic Scheduling algorithms that work well for aperiodic and periodic tasks, respectively. Note that these algorithms do not take resource locking into account. The moment we consider locks, it is possible that a low-priority task holds a lock that is required by a high-priority task. This scenario is known as *priority inversion*. There are many algorithms in this space that reduce the likelihood of repeated (and in some cases unbounded) priority inversion. We shall conclude this chapter with discussing such protocols and their trade-offs.

5.1 Synchronization

5.1.1 Data Races

Consider the case of a multicore CPU. We want to do a very simple operation, which is to just increment the value of the `count` variable that is stored in memory. It is a regular variable and incrementing it should be easy. Listing 5.1 shows that it translates to three assembly-level instructions. We are showing C-like code without the semicolon for the sake of enhancing readability. Note that each line corresponds to one line of assembly code (or one machine instruction) in this code snippet. `count` is a global variable that can be shared across threads. `t1` corresponds to a register (private to each thread and core). The first instruction loads the variable `count` to a register, the second line increments the value in `t1` and the third line stores the incremented value in the memory location corresponding to `count`.

Listing 5.1: Assembly code corresponding to the `count++` operation

```
t1 = count
t1 = t1 + 1
count = t1
```

This code is very simple, but when we consider multiple threads, it turns out to be quite erroneous because we can have several correctness problems. Consider the scenario shown in Figure 5.2. Note again that we first load the value into a register, then we increment the contents of the register and finally save the contents of the register in the memory address corresponding to the

variable `count`. This makes a total of 3 instructions that are not executed atomically; they can be executed at three different instants of time. Here there is a possibility of multiple threads trying to execute the same code snippet at the same point of time and also updating `count` concurrently. This situation is called a *data race* (a more precise and detailed definition follows later).
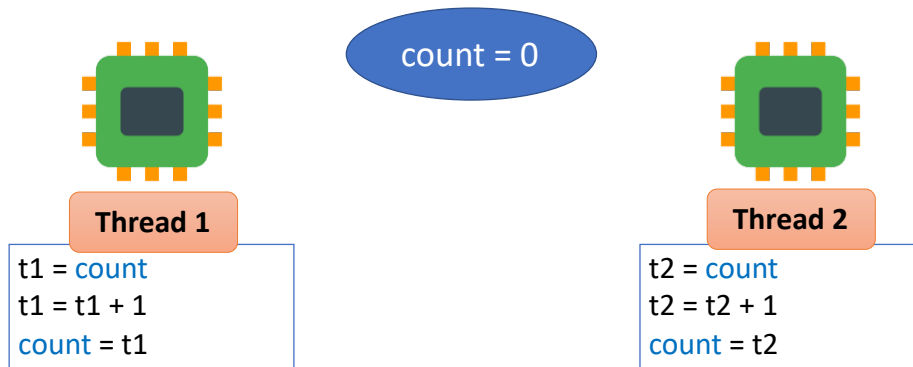

Figure 5.2: Incrementing the `count` variable in parallel (two threads). The run on two different cores. `t1` and `t2` are thread-specific variables mapped to registers

Before we proceed towards that and elaborate on how and why a data race can be a problem, we need to list a couple of assumptions.

① The first assumption is that each basic statement in Listing 5.1 corresponds to one line of assembly code, which is assumed to execute *atomically*. This means that it appears to execute at a single instant of time.

② The second assumption here is that the delay between two instructions can be indefinitely long (arbitrarily large). This could be because of hardware-level delays or could be because there is a context switch and then the context is restored after a long time. We cannot thus assume anything about the timing of the instructions, especially the timing between consecutive instructions given that there could be indefinite delays for the aforementioned reasons.

Now given these assumptions, let us look at the example shown in Figure 5.2 and one possible execution in Figure 5.3. Note that a parallel program can have many possible executions. We are showing one of them, which is particularly problematic. We see that the two threads read the value of the variable `count` at exactly the same point of time without any synchronization or coordination between them. Then they store the value of the `count` variable in two registers (temporary variables `t1` and `t2`, respectively). Finally, they increment their respective registers, and then store the incremented values in the memory address corresponding to `count`. Since we are calling the instruction `count++` twice, we expect the final value of `count` to be equal to 2 (recall that it was initialized to 0).

In this example, we get to see that the final value of `count` is equal to 1, which is clearly incorrect. Basically because there was a competition or a data race between the threads, the value of `count` could not be incremented correctly. This allowed both the threads to compete or race, which did not turn out to be a good idea in hindsight. Instead, we should have allowed one thread to complete
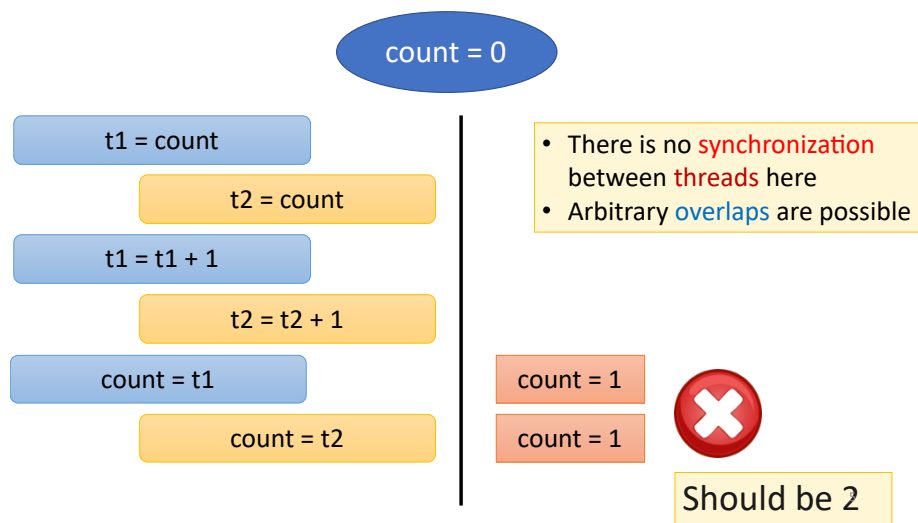
Figure 5.3: An execution that leads to the wrong value of the `count` variable

the entire sequence of operations first, and then allowed the other thread to do the same. The final value of `count` would have been correctly set to 2.

The main issue here is that of competition between threads. The overlapped execution does not lead to an intuitive outcome. Hence, there is a need for a *locking* mechanism that sequentializes the execution. A lock needs to be acquired before we enter such a sequence of code, which is also referred to as a *critical section*. At any point of time, a lock can only be acquired by one thread. It needs to be *released* before another thread can acquire it. In other words, if multiple threads try to acquire the lock at the same time, then only one of them is successful. This successful thread proceeds to execute the instructions in the critical section, which in this case increments the value of the shared variable `count`. Finally, there is a need to release the lock or unlock it. Once this is done, the other threads waiting to acquire the lock can compete among themselves to acquire it. The same process continues. Any thread that has acquired the lock can immediately begin to execute the associated critical section.

This is how traditional code works using locks; this mechanism is extremely popular and effective. All shared variables such as `count` should always be accessed using such kind of lock-unlock mechanisms. This mechanism avoids such competing situations because locks play the role of access synchronizers (see Figure 5.4).

Figure 5.5 shows the execution of the code snippet `count++` by two threads. Note the critical sections, the use of the lock and unlock calls. Given that the critical section is protected with locks, there are no data races here. The final value is correct: `count = 2`.
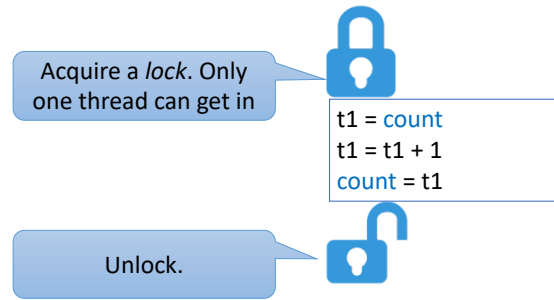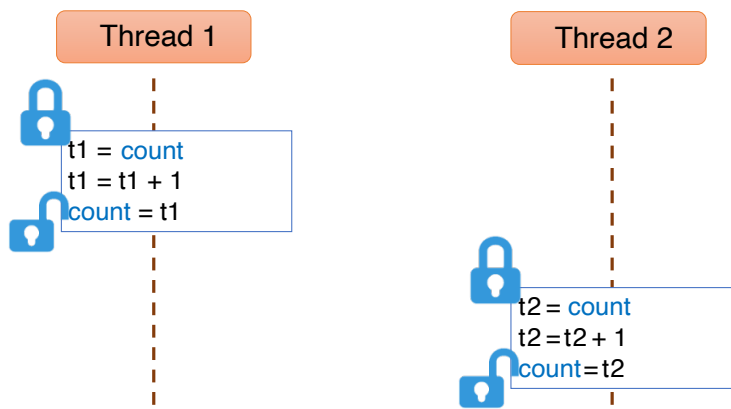
Figure 5.4: Protection of a critical section with locks

Figure 5.5: Two threads incrementing `count` by wrapping the critical section within a lock-unlock call pair

---

Definition 5.1.1 Critical Section

If the same shared variable is accessed concurrently by more than one threads and one of the accesses is a write, there is a possibility of getting non-intuitive outcomes. Such a scenario is known as a *data race* (informal definition).

To discipline such executions, such code segments (known as critical sections) should be encapsulated within a lock-unlock call pair. A lock can be acquired by only one thread at a time. This ensures that only one thread can execute the critical section at any given point of time. Critical sections cannot be executed concurrently. Subsequently, the lock needs to be released such that other threads can execute the critical section.

5.1.2   Design of a Simple Lock

Let us now look at the design of a simple lock (refer to Figure 5.6). It is referred to the test-and-test-and-set (TTAS) lock. A lock is always associated with an address. In this case, it is address A as shown in the figure. Let us use the
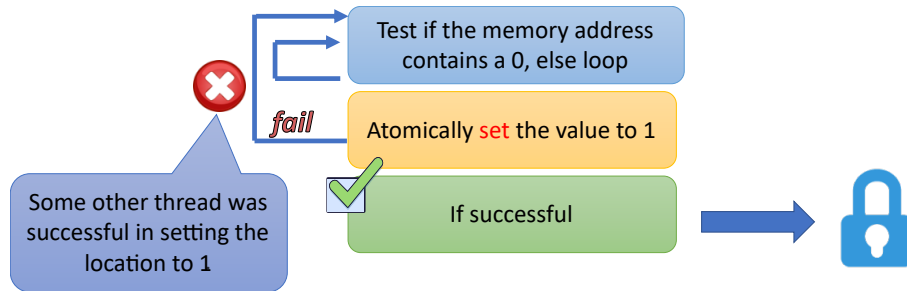
Figure 5.6: The test-and-test-and-set (TTAS) lock

convention that if the lock is free, then its value is 0 otherwise if it is busy, its value is set to 1.

All the threads that are interested in acquiring the lock need to keep checking the value stored in address A (*test* phase). If the value is equal to 1, then it means that the lock is already acquired or in other words it is busy. Once a thread finds that the value has changed back to 0 (free), it tries to set it to 1 using a special instruction (test-and-set phase). In this case, it is inevitable that there will be a competition or a race among the threads to acquire the lock (set the value in A to 1). Regular reads or writes cannot be used to implement such operations because we want the entire test-and-set process to appear to execute instantaneously (i.e., atomically).

It is important to use an atomic synchronizing instruction that almost all the processors provide, as of today. For instance, we can use the `test-and-set` instruction that is already available on most hardware. This instruction checks the value of the variable stored in memory and if it is 0, it atomically sets it to 1 (appears to happen instantaneously). If it is able to do so successfully $(0 \rightarrow 1)$, it returns a 1, else it returns 0. This basically means that if two threads are trying to set the value of a free lock variable to 1, only one of them will be successful. The hardware guarantees this feature. To summarize, the `test-and-set` instruction returns 1 if it is successful, and it returns 0 if it fails (cannot set $0 \rightarrow 1$).

We can extend this argument to n concurrent threads that all want to convert the value of the lock variable from 0 to 1. Only one of them will succeed. The thread that is successful is deemed to have *acquired* the lock. For the rest of the threads that were unsuccessful, they need to keep trying (iterating). This process is also known as *busy waiting*. Such a lock that involves busy waiting is also called a spin lock.

It is important to note that we are relying on a hardware instruction that atomically sets the value in a memory location to another value and indicates whether it was successful in doing so or not. There is a lot of theory around this and there are also a lot of hardware primitives that play the role of such atomic operations. Many of them fall in the class of read-modify-write (RMW) operations. They *read* the value stored at a memory location, sometimes test if it satisfies a certain property or not, and then they modify the contents of the memory location accordingly. These RMW operations are typically used in implementing locks. The standard method is to keep checking whether the lock

variable is free or not. The moment the lock is found to be free, threads compete to acquire the lock using atomic instructions. Atomic instructions guarantee that only one instruction is successful at a time. Once a thread acquires the lock, it can proceed to safely access the critical section. After executing the critical section, unlocking is quite simple. The lock variable needs to be set to 0 (free).

This entire process is sadly not all that simple. We have the following requirement. All the memory operations that have been performed in the critical section should be visible to all the threads running on other cores once the lock is released. This will not happen in normal circumstances since architectures and compilers tend to *reorder* instructions for performance reasons. Also, it is possible that the instructions in the critical section are visible to other threads before they observe the lock to be acquired. This is again another non-intuitive consequence of reordering. Such reordering needs to be done in a disciplined manner. Otherwise, it is not possible to correctly implement critical sections.

The Fence Instruction

Due to the aforementioned reasons, there is a need to insert a *fence* instruction whose job is to basically ensure that all the writes that have been made before the fence instruction (in program order) are visible to all the threads once the fence instruction completes. Such fence instructions are required when we perform both lock and unlock operations. A fence is also required while acquiring a lock because we need to ensure that no instruction in the critical section takes effect until the fence associated with the lock operation has completed. The critical section therefore needs to be encapsulated by fence instructions at both ends. This will ensure that the critical section executes correctly on a multicore machine. All the reads/writes are correctly visible to the rest of the threads.

> **Point 5.1.1**
>
> Fence instructions are expensive in terms of performance. Hence, we need to minimize them. They are however required to ensure correctness in multithreaded programs and to implement lock-unlock operations correctly.

This is why most atomic instructions either additionally act as fence instructions or a separate fence instruction is added by the library code to lock/unlock functions. Let us delve further and understand the theory behind reordering and fence instructions.

> **Trivia 5.1.1**
>
> Fence instructions are also known as memory barriers.

5.1.3 Theory of Data Races

We have seen examples of data races and informally understand what they are. Let us now study them more formally. A *data race* is defined as a **concurrent** and **conflicting** access to a shared variable by at least two threads. Two accesses across threads are said to be conflicting if they access the same shared

variable and one of them is a write. It is easy to visualize why this is a conflicting situation because clearly the order of the operations matters. If both the operations are read operations, then for obvious reasons, the order does not matter.

Defining concurrent accesses is slightly more difficult; it would require much more theory. We will thus only provide a semi-formal definition here. Readers are advised to read the textbook on Next-Gen Computer Architecture by your author [Sarangi, 2023]. This topic is explained in great detail. In this book, we will just provide high-level cursory details.

We need to first appreciate the notion of a *happens-before* relationship in concurrent systems. Event a is said to <u>happen before</u> event b if in a given execution, a leads to a chain of events that ultimately lead to b. Note that a happens-before relationship primarily holds in the context of a given execution of a program. In a different execution, a different happens-before relationship may hold. If a program is written in such a way that there will always be a happens-before relationship between two events regardless of the execution, then we can make a general statement of the following form: there is always a happens-before relationship between events a and b.

We can visualize a happens-before relationship in Figure 5.5, where we show how two threads execute two instances of the `count++` operation. After incrementing the `count` variable for the first time, the corresponding lock is released. There is a happens-before relationship between the update to `count` and the lock release operation. This makes intuitive sense given that the effects of a critical section should be visible before the lock is released. No update should be visible after the lock release. Next, there is a happens-before relationship between this unlock operation and the subsequent lock operation (issued by Thread 2). This is because atomic instructions with in-built fences are used to perform lock-related operations, and such memory operations are sequentially consistent with respect to each other on most architectures. Intuitively, the lock acquisition by Thread 2 should appear to happen after the lock release initiated by Thread 1. The final update to the `count` variable needs to appear to happen after the lock acquisition (by Thread 2).

Given that the happens-before relationship respects transitivity, we can say that there is a happens-before relationship between the first and second updates to `count`. The writes to `count` are thus <u>ordered</u>.

The moment we do not have such happens-before relationships between accesses, they are deemed to be concurrent. Note that in our example, such happens-before relationships are being enforced by the lock/unlock operations and their inherent fences. Happens-before order: updates in the critical section of Thread 1 → unlock operation in Thread 1 → lock operation in Thread 2 → reads/writes in the second critical section (Thread 2). Encapsulating critical sections within lock-unlock pairs creates such happens-before relationships. Otherwise, we have data races.

Such data races are clearly undesirable as we saw in the case of `count++`. Hence, concurrent and conflicting accesses to the same shared variable should not be there. With data races, it is possible that we may have hard-to-detect bugs in the program. Also, data races have a much deeper significance in terms of the correctness of the execution of parallel programs. At this point we are not in a position to appreciate all of this. All that can be said is that data-race-free programs have a lot of nice and useful properties, which are very important

in ensuring the correctness of parallel programs. Hence, data races should be avoided for a wide variety of reasons. Refer to the book by your author on Advanced Computer Architecture [Sarangi, 2023] for a detailed explanation of data races, and their implications and advantages. We shall discuss the importance of data-race-free programs later on. Theorem 5.1.6 states that if a parallel execution is data-race-free, then we can reason about it in terms of its equivalent sequential execution.

> **Point 5.1.2**
>
> An astute reader may argue that there have to be data races on the lock variable to acquire the lock itself. However, those use atomic instructions and happen in a very controlled manner; hence, they don't pose a correctness problem. This part of the code is heavily verified and is provably correct. The same cannot be said about data races in regular programs that involve regular variables.

Properly-Labeled Programs

Now, to avoid data races, it is important to create properly labeled programs. In a properly labeled program, the same shared variable should be locked by the same lock or the same set of locks. This will avoid concurrent accesses to the same shared variable. For example, the situation shown in Figure 5.7 has a data race on the variable C because it is not protected by the same lock in both the cases. Hence, we may observe a data race because accesses to C are not adequately protected. This is why it is important that we ensure that the same variable is protected by the same lock (could also be a set of multiple locks).
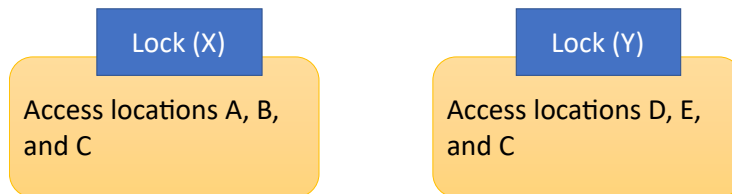

Figure 5.7: A figure showing an execution with two critical sections. The first is protected by lock X and the second is protected by lock Y. Address C is common to both the critical sections. There may be a data race on address C.

5.1.4 Deadlocks

Using locks sadly does not come for free; they can lead to a situation known as *deadlocks*. A deadlock is defined as a situation where one thread is waiting on another thread, that thread is waiting on another thread, so on and so forth – we have a cyclic wait situation. This basically means that in a deadlocked situation, no thread can make any progress. In Figure 5.8, we show such a situation with locks.

It shows that one thread holds lock X, and it tries to acquire lock Y. On the other hand, the second thread holds lock Y and tries to acquire lock X. There
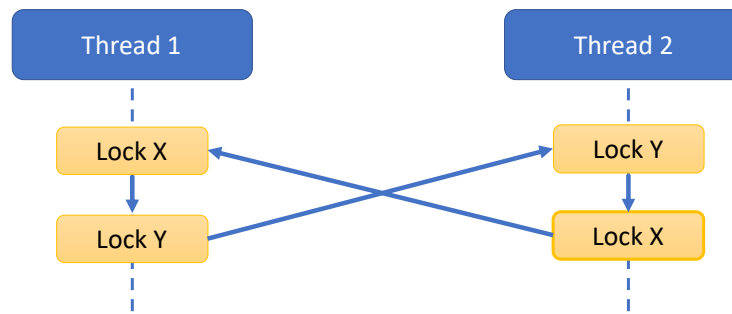
Figure 5.8: A situation with deadlocks (two threads)

is a clear deadlock situation here. It is not possible for any thread to make progress because they are waiting on each other. This is happening because we are using locks and a thread cannot make any progress unless it acquires the lock that it is waiting for. A code with locks may thus lead to such kind of deadlocks that are characterized by circular waits. Let us elaborate further by looking at the precise conditions that lead to a deadlock.

There are four conditions for a deadlock to happen. This is why if a deadlock is supposed to be avoided or prevented, one of these conditions needs to be prevented/avoided. The precise conditions are as follows:

1. **Hold-and-wait**: In this case, a thread holds on to a set of locks and waits to acquire another lock. We can clearly see this happening in Figure 5.8, where we are holding on to a lock and trying to grab one more lock.

2. **No preemption**: It basically means that a lock cannot be forcibly taken away from a thread after it has acquired it. This follows from the literal meaning of the word "preemption", which basically means taking away a resource from a thread that has already acquired it. In general, we do not preempt locks. For instance, we are not taking away lock X from thread 1 to avoid a potential deadlock situation (see Figure 5.8).

3. **Mutual exclusion**: This is something that follows directly from the common sense definition of a lock. It basically means that a lock cannot be held by two threads at the same time.

4. **Circular wait**: As we can see in Figure 5.8, all the threads are waiting on each other and there is a circular or cyclic wait. A cyclic wait ensures that no thread can make any progress.

The Dining Philosopher's Problem

In this context, the Dining Philosopher's problem is very important. Refer to Figure 5.9, which shows a group of philosophers sitting on a circular table. Each philosopher has two forks on his left and right sides. He can only pick one fork at a time. A philosopher needs both the forks to start his dinner. It is clear that this scenario involves something that we have seen in locking. Picking a fork basically means locking it and proceeding with both the forks (left and right
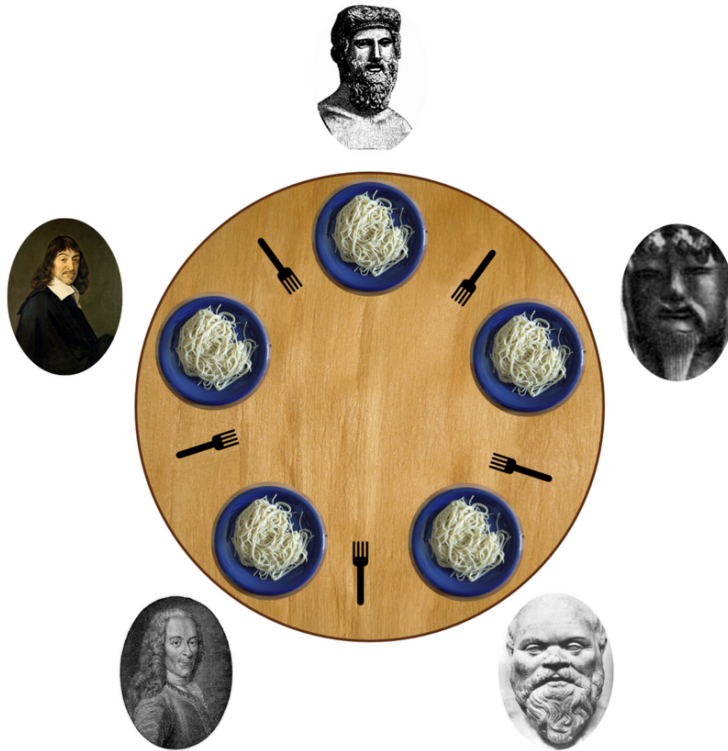
Figure 5.9: The Dining Philosopher's problem (source: Wikipedia, Benjamin D. Eshram, licensed under CC-BY-SA 3.0)

ones) and starting to eat is the same as entering the critical section. This means that both the forks have to be acquired.

It is very easy to see that a deadlock situation can form here. For instance, every philosopher can pick up his left fork first. All the philosophers can pick up their respective left forks at the same time and keep waiting for their right forks to be put on the table. These have sadly been picked up from the table by their respective neighbors. Clearly a circular wait situation has been created. Let us look at the rest of the deadlock conditions, which are mutual exclusion, non-preemption and hold-and-wait, respectively. Clearly mutual exclusion will always have to hold because a fork cannot be shared between neighbors at the same point of time.

Preemption – forcibly taking away a fork from a philosopher – seems to be difficult because its neighbor can also do the same. Designing a protocol around this idea seems to be difficult. Let us try to relax hold-and-wait. The aim here is to either grab both the forks together at the same time or grab none at all. No philosopher will grab a single fork and wait for the other. The problem with this scheme is that it is not possible to guarantee that both the forks can be picked up together. This is not an atomic operation. Each philosopher still has to grab the forks one after the other. Once he grabs one fork, he need not be able to get access to the next fork. There is no guarantee of success, if the fork

is kept back on the table and the philosopher decides to retry at a later point of time.

Hence, the simplest way of dealing with this situation is to try to avoid the circular wait condition. In this case, we would like to introduce the notion of asymmetry, where we can change the rules for just one of the philosophers. Let us say that the default algorithm is that each philosopher picks his left fork first and then the right one. We change the rule for one of the philosophers: he acquires his right fork first and then the left one.

It is possible to show that a circular wait cannot form. Let us number the philosophers from 1 to n. Assume that the nth philosopher is the one that has the special privilege of picking up the forks in the reverse order (first right and then left). In this case, we need to show that a cyclic wait can never form.

Assume that a cyclic wait has formed. It means that a philosopher (other than the last one) has picked up the left fork and is waiting for the right fork to be put on the table. This is the case for philosophers 1 to $n-1$. Consider what is happening between philosophers and $n-1$ and n. The $(n-1)^{th}$ philosopher picks its left fork and waits for the right one. The fact that it is waiting basically means that the nth philosopher has picked it up. This is his left fork. It means that he has also picked up his right fork because he picks up the forks in the reverse order. Recall that he first picks up his right fork and then his left one. This basically means that the nth philosopher has acquired both the forks and is thus eating his food. He is not waiting, and therefore there is no deadlock. This leads to a contradiction. Hence, a deadlock cannot form using this protocol, where we deliberate introduce some degree of asymmetry.

Deadlock Prevention, Avoidance and Recovery

Deadlocks are clearly not desirable. Hence, as far as possible, we would like to steer clear of them. There are several strategies here. The first is that we try to prevent them such that they do not happen in the first place. This is like taking a vaccine to prevent the disease from happening. To do this, we need to ensure that at least one of the four deadlock conditions does not materialize. For example, we can design a lock acquisition protocol such that it is never possible to create a circular wait situation.

Consider the case when we know the set of locks a given operation will acquire a priori. In this case, we can follow a simple *2-phase locking protocol*. In the first phase, we simply acquire all the locks in ascending order of their addresses. Subsequently, in the second phase, we release all the locks. The key assumption is that all the locks that will be acquired are known in advance. In reality, this is not a very serious limitation because in many practical use cases, this information is often known.

The advantage here is that we will not have deadlocks. This is because a circular wait cannot happen. There is a fundamental asymmetry in the way that we are acquiring locks in the sense that we are acquiring them in ascending order of addresses.

Let us prove deadlock prevention by arriving at a contradiction. Assume that there is a circular wait. Let us annotate each edge uv in the cycle with the lock address A. In this case, Process P_u wants to acquire lock A that is currently held by P_v. As we traverse this list of locks along the cycle, addresses will continue to increase because a process always waits on a lock whose address

is larger than the address of any lock that it currently holds. Continuing on these lines, we observe that in a circular wait, the lock addresses shall keep increasing. Given that there is a circular wait, there will be a process P_x that is waiting for lock A that is held by P_y ($P_x \rightarrow P_y$). Given the circular wait, assume that P_x holds lock A', which P_z is waiting to acquire ($P_z \rightarrow P_x$). We have a circular wait of the form $P_x \rightarrow P_y \rightarrow \ldots \rightarrow P_z \rightarrow P_x$. Now, lock addresses need to increase as we traverse the cycle. This is because a process always covets a lock whose address is higher than the addresses of all the locks that it currently holds (due to the two-phase locking protocol). We thus have $A' > A$. Now, P_x holds A', and it waits for A. This means that $A > A'$. Both cannot be true. We thus have a contradiction. Hence, a circular wait is not possible.

The third approach is deadlock avoidance. This is more like taking a medicine for a disease. In this case, before acquiring a lock, we check if a deadlock will happen or not, and if there is a possibility of a deadlock, then we do not acquire the lock. We throw an exception such that the user process that initiated the lock acquisition process can catch it and take appropriate action.

The last approach is called *deadlock recovery*. Here, we run the system optimistically. We have a deadlock detector that runs as a separate thread. Whenever, we detect sustained inactivity in the system, the deadlock detector looks at all the shared resources and tries to find cycles. A cycle may indicate a deadlock (subject to the other three conditions). If such a deadlock is detected, there is a need to break it. Often sledgehammer like approaches are used. This means either killing processes or forcefully taking locks away from them.

Starvation and Livelocks

Along with deadlocks, there are two more important issues that need to be discussed namely *starvation* and *livelocks*. Starvation means that a thread tries to complete an operation such as acquire a lock but fails to do so for an indefinite period. This means that it participates in the race to acquire the lock by atomically trying to convert the value of a memory location from free to busy. However, it loses all the time. There is no guarantee of success and thus it can end up trying for an indefinite period. Therefore, it may have to wait forever for the desired operation to complete.

This is clearly a very important problem, and it is thus necessary to write elaborate software libraries using native atomic hardware primitives that prevent starvation. The algorithm should be designed in such a way that no thread has to wait infinitely for its operation to complete. *Starvation freedom* is indeed a very desirable property because it indicates that within a finite (in some cases bounded) amount of time, a thread completes its operation. In the case of a lock, it either gets access to some resource or gets to execute the critical section. Note that starvation freedom also implies deadlock freedom because it would not allow processes to deadlock and wait forever. However, the converse is not true. Deadlock freedom does not imply starvation freedom because starvation is a much stronger condition.

The other condition is a livelock, where processes continuously take steps and execute statements but do not make any tangible progress. This means that even if processes continually change their state, they do not reach the final end state – they continually cycle between interim states. Note that they are not in a deadlock in the sense that they can still take some steps and keep changing their

state. However, the states do not converge to the final state, which indicates a desirable outcome. In older Ethernet networks, livelocks were quite common. A single channel was shared between different machines. If two machines tried to transmit together at the same time, then there could be a collision and the machines had to back off for a random duration, and transmit again. This process did not guarantee successful message transmission. The messages could collide again and again. This is an example of a livelock because there is visible progress in terms of internal states changing and messages being sent. However, there is no successful message transmission.

Consider another quintessential example. Two people are trying to cross each other in a narrow corridor. A person can either be on the left side or on the right side of the corridor. So it is possible that both are on the left side, and they see each other face to face. Hence, they cannot cross each other. Then they decide to either stay there or move to the right. It is possible that both of them move to the right side at the same point of time, and they are again face to face. Again they cannot cross each other. This process can continue indefinitely. In this case, the two people can keep moving from left to right and back. However, they are not making any progress because they are not able to cross each other. This situation is a livelock, where threads move in terms of changing states, but nothing useful gets ultimately done.

> **Point 5.1.3**
>
> Starvation freedom ensures freedom from deadlocks and livelocks. However, freedom from deadlocks and livelocks does not guarantee starvation freedom.

5.1.5 Pthreads and Synchronization Primitives

Let us now look at *pthreads* or Posix threads, which is the most popular way of creating threads in Linux-like operating systems. Many other thread APIs use pthreads as their base. The code for creating pthreads is shown in Listing 5.2. We wish to execute the function `foo` in parallel.

Note the signature of `foo`. It takes a generic `void *` pointer as its sole argument and also returns a `void *` pointer. The rationale behind this is quite clear. We want pthreads to be a generic mechanism. It should be possible to run any function concurrently on a separate thread. We are thus not sure about its arguments and return value. Hence, it is a good idea to pass a pointer to an argument or a structure (in case there are multiple arguments). Similarly, it is a good idea to just return just a generic `void *` pointer, which can point to anything and can also be NULL. This design choice ensures that the function executed by a pthread is generic in character. In the case of the `foo` function, we extract the argument (thread id) and we simply print it. Subsequently, we multiply the argument with 2 and return the product.

Listing 5.2: Code to create two pthreads and collect their return values

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
```

```c
pthread_t tid[2];
int count;
void* foo(void *arg) {
    int *ptr = (int *) arg;  /* get the argument: thread id
        */
    printf("Thread %d \n", *ptr);  /* print the thread id */

    /* send a custom return value */
    int *retval = (int *) malloc (sizeof(int));
    *retval =  (*ptr) * 2; /* return 2 * thread_id */
}

int main(void) {
    int errcode, i = 0; int *ptr;

    /* Create two pthreads */
    for (i=0; i < 2; i++) {
        ptr = (int *) malloc (sizeof(int));
        *ptr = i;
        errcode = pthread_create(&(tid[i]), NULL,
                        &foo, ptr);
        if (errcode)
            printf("Error in creating pthreads \n");
    }

    /* Wait for the two pthreads to finish and join */
    int *result;
    pthread_join(tid[0], (void **) &result);
    printf ("For thread 0, %d was returned \n", *result);

    pthread_join(tid[1], (void **) &result);
    printf ("For thread 1, %d was returned \n", *result);
}
```

Let us now discuss the `main` function that creates two pthreads. The arguments to the `pthread_create` function are a pointer to the pthread structure, a pointer to a pthread attribute structure that controls its behavior (NULL in this example), the pointer to the function that needs to be executed and a pointer to its sole argument. If the function takes multiple arguments, then we need to put all of them in a structure and pass a pointer to that structure.

In our example, the return value of the `foo` function is a pointer to an integer that is equal to 2 times the thread id. When a pthread function (like `foo`) returns, akin to a signal handler, it returns to the address of a special routine. This routine does the job of cleaning up the state and destroying the thread. Once the thread finishes, the parent thread that spawned it can wait for it to finish using the `pthread_join` call.

This is similar to the `wait` call invoked by a parent process, when it waits for a child to terminate in the regular fork-exec model. In the case of a regular process, we collect the exit code of the child process. However, in the case of pthreads, the `pthread_join` call takes two arguments: the pthread, and the address of a pointer variable (`&result`). The value filled in the address (value of `result`) is the pointer that the corresponding pthread function instance re-

turned. We can proceed to dereference the pointer `result` and extract the value that the function wanted to return.

Given that we have now created a mechanism to create pthread functions that can be made to run in parallel, let us implement a few concurrent algorithms.

> **Trivia 5.1.2**
>
> To compile a piece of code that uses pthreads, we need to use the command `gcc prog_name -lpthread`.

Incrementing a Shared Variable using Lock and Unlock Calls

Listing 5.3: Lock-unlock using pthreads

```c
int count = 0;
pthread_mutex_t cntlock;  /* the lock variable */

void* func(void *arg) {
    pthread_mutex_lock(&cntlock);     /* lock */
    count++;
    pthread_mutex_unlock(&cntlock);   /* unlock */
}
int main () {
    retval = pthread_mutex_init (&cntlock, NULL);
    ...
    ...
    printf ("The final value of count is %d \n", count);
}
```

Consider the code shown in Listing 5.3. A lock in pthreads is of type `pthread_mutex_t`. It needs to be initialized using the `pthread_mutex_init` call. The first argument is a pointer to the pthread mutex (lock), and the second argument is a pointer to a pthread attributes structure. If it is NULL, then it means that the lock will exhibit its default behavior.

The lock and unlock functions are indeed quite simple here. We can just use the calls `pthread_mutex_lock` and `pthread_mutex_unlock`, respectively. All the code between them comprises the critical section. In this case, we are just incrementing the value of the variable `count` in the critical section.

Incrementing a Shared Variable without Using Locks

Listing 5.4: Lock-unlock using atomic fetch and add

```c
#include <stdatomic.h>
atomic_int count = 0;

void * fetch_and_increment (void *arg) {
    atomic_fetch_add (&count, 1);
}
```

Next, let us use atomics that wrap hardware-level atomic instructions. Atomics, in their current form, were originally defined in C++ 11. Most processors, provide an atomic version of the fetch and add instruction that is guaranteed to complete in a bounded amount of time. x86 processors provide such an instruction. All that needs to be done is to add the `lock` prefix in front of an add instruction – it becomes an *atomic add* instruction.

The `fetch_and_increment` function makes a call to the C++ `atomic_fetch_add` instruction on x86 processors. This instruction appears to execute instantaneously and completes within a bounded amount of time.

This is a classic example of a non-blocking algorithm that does not use locks. It also belongs to the class of lock-free algorithms. Such algorithms are clearly way better than variants that use locks.

Let us look at another example that uses a different atomic primitive – the compare-and-swap (CAS) instruction.

Incrementing a Shared Variable using the CAS Library Function

In C++, the `atomic_compare_exchange_strong` method is normally used to implement the classic compare and swap operation. It is typically referred to as the CAS operation. The standard format of this method is as follows: `CAS(&val,&old,new)`. The logic is as follows. If the comparison is successful (`val==old`), then `val` is set equal to `new`. Given that we are passing a pointer to `val`, the value of `val` can be modified within this function. If they are not equal (`val ≠ old`), then `old` is set equal to the value of `val`. The pseudocode of this method is shown in Listing 5.5. Note that the entire method executes atomically using x86's `cmpxchg` instruction.

Listing 5.5: The operation of the CAS method in C-like code

```
bool CAS (int *valptr, int *oldptr, int new) {
    if (*valptr == *oldptr) {   /* equality */
        *valptr = new;          /* set the new value */
        return true;
    } else {                    /* not equal */
        *oldptr = *valptr;      /* old = val */
        return false;
    }
}
```

Let us now use the CAS method to increment `count` (code shown in Listing 5.6).

Listing 5.6: Lock-unlock using the compare and swap instruction

```
atomic_int count = 0;
#define CAS atomic_compare_exchange_strong

void* fetch_and_increment (void *arg) {
    int oldval, newval;
    do {
        oldval = atomic_load (&count);
        newval = oldval + 1;
        printf ("old = %d, new = %d \n", oldval, newval);
    } while (!CAS (&count, &oldval, newval));
```

```
}
```

The `fetch_and_increment` function is meant to be called in parallel by multiple pthreads. We first load the value of the shared variable `count` into `oldval`, next compute the incremented value `newval`, and then try to atomically set the value of `count` to `newval` as long as its value is found to be equal to `oldval`. This part is done atomically. If the CAS operation is not successful because another thread was able to update the value at the same time, then **false** will be returned. There is thus a need to keep iterating and trying again and again until the CAS operation is successful. Note that there is no guaranteed termination in this algorithm. In theory, a thread can starve and keep losing the CAS (getting a **false**) forever.

Now that we have looked at various methods of incrementing a simple count variable, let us delve deeper into this. Let us understand the theory of concurrent non-blocking algorithms.

5.1.6 Theory of Concurrent Programs

Let us consider all the examples of codes that do not use locks. As discussed before, they are known as non-blocking algorithms. Clearly they are a better choice than having locks. However, as we have discussed, such algorithms are associated with their fare share of problems. They are hard to write and verify. Hence, we should use non-blocking algorithms whenever we find simple ones available. The additional complexity often justifies the resultant performance benefits.

Correctness Criteria

Let us now proceed to formally define what a concurrent program is and what are its correctness guarantees. We can either have variants that use locks or variants that do not use locks such as the examples that we saw earlier. Recall that they used the compare-and-swap or fetch-and-add primitives. This class of programs or algorithms are known as non-blocking programs (or algorithms). It turns out that there are numerous types of non-blocking algorithms. They can be classified into broadly three different classes based on the progress guarantees that they make: obstruction-free, lock-free and wait-free. Before delving into this, let us start with some basic terminology and definitions.

A concurrent algorithm has a set of well-defined operations that change the global state – set of all the shared variables visible to all the threads. For example, we can define a concurrent algorithm to operate on a shared global queue. The *operations* can be *enqueue* and *dequeue*, respectively. Similarly, we can define operations on a concurrent stack that execute concurrently. Each such operation has a *start* and *end* time, respectively. These are distinct and discrete points of time. The start of an operation is a *point of time* when the corresponding method is invoked. The end of an operation is when the method (function) returns. If it is a read or an operation like a read that does not modify the state, then the method returns with a value. Otherwise, the method achieves something similar to a write operation (change of state). The method in this case does not return with a value, it instead returns with a value indicating the status of the operation: success, failure, etc. In this case, we do not know when

the method actually takes effect. It can make changes to the underlying data structure (part of the global state) before the end of the method, or sometime after it as well. This difference is quite **fundamental**.

Atomicity

If the method appears to take effect *instantaneously* at a unique point of time, then it is said to be *atomic*. The key word over here is atomic. Regardless of the way that a method actually executes, it should appear to any external observer that it has executed *atomically*. This means that an external entity cannot observe any intermediate state – it either perceives the method to have fully executed or not started at all. For many readers, this may appear to be non-intuitive. After all, how can a large method that may require hundreds of instructions appear to execute in one instant? The answer is that the large method is not executing in an infinitesimally small instant. Instead, it is "appearing" to execute in an *instant*. Both are different. Loosely speaking, it should not be possible for any thread to observe the state created by a partially executed method. This means that with every atomic method, we can associate a distinct point of completion (execution). Before this point of time is reached, it should appear to other threads that this method has never executed and after this point, it should appear to all the threads that the method has fully completed. A parallel execution is said to be atomic if all the methods in it appear to execute atomically. We need to understand that this is a theoretical definition. The entire method is obviously not executing in entirety at the point of completion.

Let us assume that we somehow know these completion points (may not be the case in practice). If we can arrange all these completion points in ascending order of physical time, then we can arrange all the methods sequentially (across threads). If we think about it, this is a way of mapping a parallel execution to a sequential execution, as we can see in Figure 5.10. Here, we are mapping a parallel execution of a concurrent queue to a sequential execution. In the sequential timeline, the methods are arranged in ascending order of their completion times. This *mapped* sequential execution is of great value because the human mind finds it very easy to reason about sequential executions, whereas it is very difficult to make sense of parallel executions. Let us say that the sequential execution (shown at the bottom of the figure) is *equivalent* to the parallel execution.

Legal Sequential Execution

If the equivalent sequential execution satisfies the semantics of the algorithm, it is said to be *legal*. For example, in Figure 5.10, we show a set of enqueue and dequeue operations that are issued by multiple threads. The parallel execution is hard to reason about (prove or disprove correctness, either way); however, the equivalent sequential execution can easily be checked to see if it follows the semantics of a queue – it needs to show FIFO behavior. Atomicity and the notion of a point of completion allow us to check a parallel algorithm for correctness. But, we are not fully there yet. We need a few more definitions and concepts in place.

The key question that needs to be answered is about the location of this *point of completion* vis-á-vis the start and end points. If it always lies between them, then we can always *claim* that before a method call ends, it is deemed to have fully completed – its changes to the global state are visible to all the
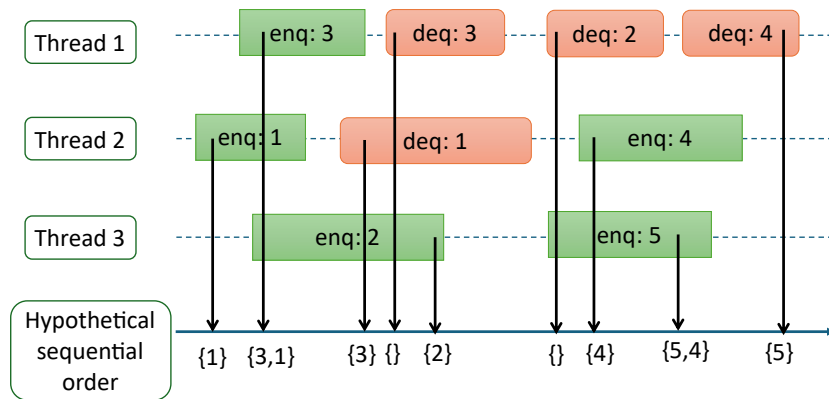
Figure 5.10: A parallel execution and its equivalent sequential execution. Every event has a distinct start time and end time. In this figure, we assume that we know the completion time of each operation. We arrange all the events in ascending order of their completion times in a hypothetical sequential order at the bottom. Each point in the sequential order shows the contents of the queue after the respective operation has completed. Note that the terminology *enq: 3* means that we enqueue 3, and similarly *deq: 4* means that we dequeue 4.

threads. This is a very strong correctness criterion of a parallel execution. The default assumption here is that the equivalent sequential execution is *legal*. This correctness criteria is known as linearizability.

Linearizability

Linearizability is the de facto criterion used to prove the correctness of concurrent data structures that are of a non-blocking nature. If all the executions corresponding to a concurrent algorithm are linearizable, then the algorithm itself is said to satisfy linearizability. For example, the execution shown in Figure 5.10 is linearizable.

This notion of linearizability is summarized in Definition 5.1.2. Note that the term "physical time" in the definition refers to real time that we read off a wall clock. Later on, while discussing progress guarantees, we will see that the notion of physical time has limited utility. We alternatively prefer to use the notion of logical time instead, which is based on the order of operations. Nevertheless, let us stick to physical time for the time being.

> **Definition 5.1.2** Linearizability
>
> An execution is said to be *linearizable* if every method call is associated with a distinct point of completion that is between its start and end points (in terms of physical time). Moreover, the equivalent sequential order is legal.

Next, let us address the last conundrum. Even if the completion times are not known, which is often the case, as long as we can show that distinct completion points *appear* to exist for each method (between its start and end), the execution is deemed to be linearizable. Mere *existence* of completion points is what needs to be shown. Whether the method actually completes at that point or not

is important. This is why we keep using the word "appears" throughout the definitions.
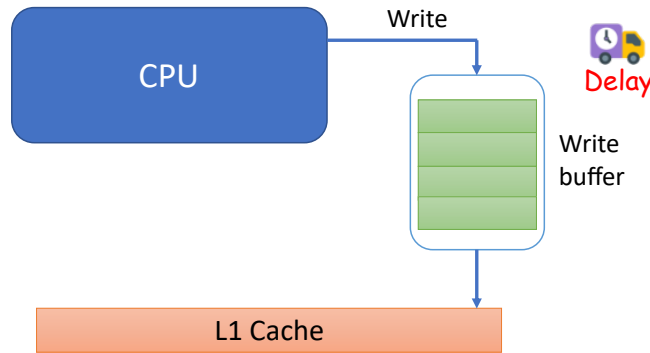
Notion of Memory Models

Figure 5.11: A CPU with a write buffer

Now consider the other case when the point of completion may be after the end of a method. For obvious reasons, it cannot be before the start point of a method. An example of such an execution, which is clearly atomic but not linearizable, is a simple write operation in multicore processors (see Figure 5.11). The write method returns when the processor has completed the write operation and has written it to its write buffer. This is also when the write operation is removed the pipeline. However, this does not mean that the write operation has *completed*. It completes when it is visible to all the threads, which can happen much later – when the write operation leaves the write buffer and is written to a shared cache. This is thus a case when the completion time is beyond the end time of the method. The word "beyond" is being used in the sense that it is "after" the end time in terms of the real physical time.

We now enter a world of possibilities. Let us once again consider simple read and write operations that are issued by cores in a multicore system. The moment we consider non-linearizable executions, the completion time becomes very **important**. The reason for preferring non-linearizable executions is that a host of performance-enhancing optimizations in the compiler, processor and memory system can be realized. One such example is the use of a write buffer, as we have just seen. Most of these optimizations involve delaying and **reordering** instructions. As a result, the completion time can be well beyond the end time, especially for writes (operations that involve a state change). Hence, the more relaxed we are in setting the completion time, higher is the performance.

The question that naturally arises is how do we guarantee the correctness of algorithms in such settings? In the case of linearizability, it was easy to prove correctness. We just had to show that for each method a point of completion exists, and if we arrange these points in ascending order of completion times, then the sequence is *legal*. Hence, for complex concurrent data structures such as stacks and queues, linearizability is preferred. However, for simpler operations like memory reads and writes, linearizability is too expensive in terms of

performance. There is a need to wait for the write operations to fully complete, which might take a lot of time.

Hence, other models are used. These models basically define the set of allowed outcomes of a parallel program. Specifically, a model defines what a read can return, when writes take effect and the behavior of synchronization operations. Such models are known as *memory models* or *memory consistency models*. The word "consistency" arises from the fact that every model needs to be <u>consistent</u> with specifications. Every multicore processor as of today defines a memory model and concomitant specifications. A specification is like a contract between hardware and software. It precisely lays down the rules governing memory and synchronization operations. The hardware guarantees certain orderings between memory operations. As long as software is written in a manner that does not run afoul of the memory model, correctness guarantees can be made.

Memory models typically confine themselves to reads, writes, atomic memory operations with built-in fences such as compare-and-swap and synchronization operations (refer to Section 5.1.2). Often there is a need to decide if a given parallel execution adheres to a given memory model or not. Answering this question is beyond the scope of this book. The textbook on Next-Generation Computer Architecture [Sarangi, 2023] by your author is the right point to start.

> **Definition 5.1.3** Memory Models
>
> A memory model or a memory consistency model is a specification for a multicore processor. It governs the behavior of reads, writes, atomic and synchronization operations. Specifically, it specifies the ways in which a core or the memory system can reorder memory operations and whether operations are atomic or not. Moreover, it specifies the behavior of synchronization operations and the additional orderings that they impose. It determines the set of valid outcomes of a parallel program.

In this context, let us first describe a memory model that is considered to be a gold standard in the world of memory consistency models. It is sequential consistency (abbreviated as SC). It is perceived to be quite slow in practice and thus not used. However, it plays a vital role in ensuring the correctness of executions.

Sequential Consistency

Sequential consistency is slightly weaker than linearizability. This means that it allows some outcomes that linearizability does not. It requires atomicity – unique completion times for each operation that are globally respected. Along with atomicity, SC mandates that in the equivalent sequential order of events, methods invoked by the same thread appear in *program order*. The program order is the order of instructions in the program that will be perceived by a single-cycle processor, which will pick an instruction, execute it completely, proceed to the next instruction, so on and so forth. SC is basically atomicity + intra-thread program order. Linearizability had an additional requirement, which stated that the point of completion should be between the start time and end time. This requirement is not there in SC.

Consider the following execution. Assume that x and y are global variables that are initialized to 0. t1 and t2 are *local* variables. They are stored in registers (not shared across threads).

Thread 1	Thread 2
x = 1	y = 1
t1 = y	t2 = x

Note that if we run this code many times on a multicore machine, we shall see different outcomes. It is possible that Thread 1 executes first and completes both of its instructions and then Thread 2 is scheduled on another core, or vice versa. Their execution can also be interleaved. Regardless of the thread scheduling policy, we will never observe the outcome t1 = t2 = 0 if the memory model is SC or linearizability. The reason is straightforward. All SC and linearizable executions respect the per-thread order of instructions. In this case, the first instruction to complete will either be x = 1 or y = 1. Hence, at least one of t1 or t2 must be non-zero.

> **Definition 5.1.4** Sequential Consistency
>
> A sequentially consistent memory model has two components: atomicity and per-thread ordering. Atomicity means that every operation appears to execute instantaneously at its completion point. Unlike linearizability, this point of completion need not be between the start time and end time. Next, consider operations A and B that are issued by the same thread and A precedes B. They need to appear in the same order in the equivalent sequential execution.

Weak Memory Models

On any real machine including x86 and ARM machines, the outcome t1 = t2 = 0 will indeed be visible because the compiler can reorder instructions that access different addresses and so can the hardware. This reordering is done to enhance performance. For executing a single thread, reordering does not matter. It will never change the final outcome. However, the moment shared variables and multiple threads enter the picture, the world changes. t1 = t2 = 0 becomes a valid outcome. This is because most modern out-of-order processors allow a later read to a different address precede an earlier write operation. Write operations typically have to wait till all the previous instructions in the pipeline are confirmed to be on the correct path and there is no possibility of a branch misprediction.

A modern memory model specifies a lot of rules with regard to which pairs of instructions can be reordered and also by whom: the hardware or the compiler. These rules can be quite complex. They are said to be *weaker* than SC because they are much more flexible in terms of the reorderings that they allow. Many also relax the requirement of atomicity – a method may be associated with multiple completion times as perceived by different threads. All such memory models are said to be *weak memory models*.

> **Point 5.1.4**
>
> All linearizable executions are also sequentially consistent. All sequentially consistent executions also satisfy the requirements of weak memory models. Note that the converse is not true.

Fences, Memory Barriers and Relaxed Consistency

Recall that we had discussed fences (also referred to as memory barriers) in Point 5.1.2 (Section 5.1.2). They can be understood better in the context of the current discussion. They basically stop reordering. A fence ensures that all the instructions before it – in the same thread and in program order – complete before it completes. It also ensures that no instruction after it in program order (in the same thread) appears to take effect (or complete) before it completes.

They are particularly important in the context of locks. This is because there is a very important theorem in computer architecture, which basically says that if all shared memory accesses are wrapped in critical sections and the program is properly labeled – the same variable is always protected by the same lock (or set of locks) – then the execution is *sequentially consistent*. This is true regardless of the underlying memory model (refer to [Sarangi, 2023]). We can now understand why creating critical sections is so important. It is because we need not bother about the memory model or what the compiler or hardware do in terms of reordering. All that we do is properly label the program.

> **Theorem 5.1.1** Data-Race-Free Programs have SC Executions
>
> A data-race-free program has a sequentially consistent execution on all machines. A program can be made data-race-free by properly labeling it. All shared variables need to be encapsulated in critical sections and a shared address should always be protected by the same lock (or same set of locks).

A more nuanced definition is captured in the RC (relaxed consistency) memory model. It defines two types of fence operations: *acquire* and *release*. The *acquire* operation corresponds to a *lock acquire*. It mandates the following: no operation after it in program order can complete unless it has completed. This makes sense. We first acquire the lock, and then we access shared variables in the critical section. The rest of the threads should see the lock being acquired first. The changes made in the critical section should succeed the lock acquisition event. Otherwise, it would be tantamount to disrespecting the semantics of the lock. Note that an *acquire* is *weaker* than a full fence. A full fence also specifies the global ordering of operations before the fence (in program order). Similarly, the *release* operation corresponds to a *lock release*. As per RC, the *release* operation can complete only if all the operations before it in program order have fully completed. Again, this also makes sense because when we release the lock, we want the rest of the threads to see all the changes that have been made in the critical section.

5.1.7 Progress Guarantees

In any concurrent system, we typically do not rely on physical time. To specify the properties of algorithms a wall clock is not used. Linearizability is the only exception. For all other consistency models, it is only the perceived order of operations that matters. We rely on a notion of causality between events where an event can be anything starting from a basic read or write operation to a more complex operation, such as an enqueue or dequeue operation on a queue. If one event leads to another event, then we say that there is a causal order or a happens-before order between them. A happens-before order either captures the flow of information or on some other HW/SW artifact that makes one operation wait for another to complete. In other words, we are looking at events logically and the logical connections between them in terms of cause-effect relationships. Furthermore, we are also assuming that between any two events, which could also be two consecutive statements in a program, the delay in terms of physical time could be indefinite. The reason for this is that there could be a context switch in the middle or there could be other hardware/device induced delays that could cause the process to get stalled for a very long time and get restored much later. Hence, it is not a good idea to rely on any sort of physical or absolute time when discussing the correctness of concurrent systems: parallel programs in common parlance.

Instead of physical time, let us use the notion of an internal step for denoting an event in a thread or a process. It is a basic action such as reading a variable, writing to a variable or executing a basic instruction. Each of these can be classified as an internal step, and we shall measure per-thread time only in terms of such internal steps. Note that internal steps in one thread have no relationship with the number of internal steps taken in another thread unless there is a causal relationship of events across threads.

In general, threads are completely independent. For example, we cannot say that if one thread executed n internal steps, the other thread would have executed m internal steps where m is some function of n. Without explicit synchronization, there should be no correlation between them. This is because we have assumed that between any two internal steps, the delay can be arbitrarily large. Hence, we are not making any assumptions about how long an internal step is in terms of absolute time. Instead, we are only focusing on the number of internal steps that a thread makes (executes), which again is unrelated to the number of internal steps that other threads take in the same time duration.

Using this notion, it is very easy to define the progress guarantees of different kinds of concurrent algorithms. Let us start with the simplest and the most relaxed progress guarantee.

Obstruction Freedom

Obstruction freedom means that in an n-thread system, if we make any subset of $(n-1)$ threads go to sleep, then the only thread that is active will be able to complete its execution in a bounded number of internal steps. This automatically means that we cannot use locks in an obstruction-free system. This is because if the thread that has acquired the lock gets swapped out or goes to sleep, no other thread can complete the operation – it will not get access to the lock.

Wait Freedom

Now, let us look at another progress guarantee, which is at the other end of the spectrum. It is known as *wait freedom*. In this case, we avoid all forms of starvation. Every thread completes its operation within a bounded number of internal steps. So in this case, starvation is not possible. The code shown in Listing 5.4 is an example of a wait-free algorithm because regardless of the number of threads and the amount of contention, it completes within a bounded number of internal steps. However, the code shown in Listing 5.6 is not a wait-free algorithm. This is because there is no guarantee that the compare and swap will be successful in a bounded number of attempts. Thus, we cannot guarantee wait freedom. However, this code is obstruction-free because if any set of $(n-1)$ threads go to sleep, then the only thread that is active will succeed in performing the CAS operation and ultimately complete the overall operation in a bounded number of steps.

Lock Freedom

Given that we have now defined what an obstruction-free and a wait-free algorithm is, we can now tackle the definition of lock freedom, which is slightly more complicated. In this case, let us count the cumulative number of steps that all the n threads in the system take. We have already mentioned that there is no correlation between the time it takes to complete an internal step across all the n threads. That remaining true, we can still take a system and count the cumulative number of internal steps taken by all the threads together. Lock freedom basically says that if this cumulative number is above a certain threshold or bound, then we can say for sure that at least one of the operations has completed successfully. Note that in this case, we are saying that at least one thread will make progress and there can be no deadlocks.

All the threads also cannot get stuck in a livelock. However, there can be starvation because we are taking a system-wide view and not a thread-specific view here. As long as one thread makes progress by completing operations, we do not care about the rest of the threads. This was not the case in wait-free algorithms. The code shown in Listing 5.6 is lock-free, but it is not wait-free. The reason is that the compare and exchange has to be successful for at least one of the threads and that thread will successfully move on to complete the increment operation. The rest of the threads will fail in that iteration. However, this is not of a great concern here because at least one thread achieves success.

Relationships between the Progress Guarantees

It is important to note that every program that is wait-free is also lock-free. This follows from the definition of lock freedom and wait freedom, respectively. If we are saying that in less than k internal steps, every thread is guaranteed to complete its operation, then in nk system-wide steps, at least one thread is guaranteed to complete its operation. By the pigeonhole principle, at least one thread must have taken k steps and completed its operation. Thus wait freedom implies lock freedom.

Similarly, every program that is lock-free is also obstruction-free, which again follows very easily from the definitions. Assume that the system as a whole takes a certain number of steps (let's say k). If $n-1$ threads in the system are

quiescent, then only one thread that is taking steps executes k steps. If k is large enough, then the sole running thread will complete its execution. Hence, the program is also obstruction-free.
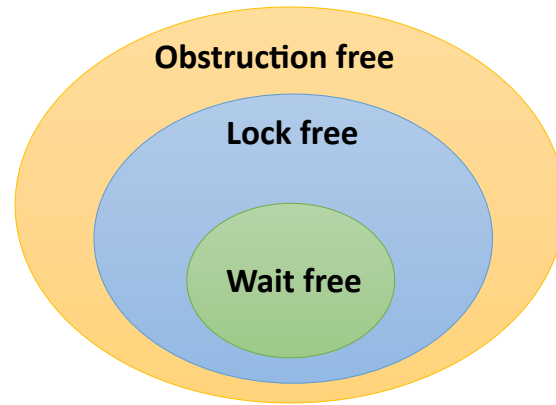

Figure 5.12: Venn diagram showing the relationship between different progress guarantees

However, the converse is not true in the sense that it is possible to find a lock-free algorithm that is not wait-free and an obstruction-free algorithm that is not lock-free. This can be visualized in a Venn diagram as shown in Figure 5.12. Note that all of these algorithms are non-blocking in character – they do not use locks. They are thus broadly known as non-blocking algorithms, even though they provide very different kinds of progress guarantees.

An astute reader may ask why not use wait-free algorithms every time because after all there are theoretical results that say that any algorithm can be converted to a parallel wait-free variant (refer to the universal construction in [Herlihy and Shavit, 2012]). The reason is that wait-free algorithms tend to be very slow and are also very difficult to write and verify. Hence, in most practical cases, a lock-free implementation is much faster and is far easier to code and verify. In general, obstruction freedom is too weak as a progress guarantee. Thus, it is hard to find a practical system that uses an obstruction-free algorithm. In most practical systems, lock-free algorithms are used, which optimally trade off performance, correctness and complexity.

5.1.8 Semaphores

Let us now consider another synchronization primitive called a *semaphore*. We can think of it as a generalization of a lock. It is a more flexible variant of a lock, which admits more than two states. Recall that a lock has just two states: locked and unlocked.

Listing 5.7: The sem_wait operation

```
/* execute atomically */
if (count == 0)
    insert_into_wait_queue ( current\_task );
else
```

```
    count --;
```

A semaphore maintains a multivalued `count`, which always needs to be positive. A semaphore can be acquired using the `sem_wait` operation (see Listing 5.7). If the count is equal to 0, then it means that no process can acquire the semaphore – the current task is put into a wait queue that can be a part of the semaphore or implemented separately in the kernel.

However, if `count` is not equal to 0, then it is decremented by 1. This essentially indicates that the semaphore has been acquired by a process. In this case, multiple processes can acquire a semaphore. For example, if the count is equal to 5, then 5 processes can acquire the semaphore. The semaphore acquire operation is referred to as a `wait` operation or alternatively a (`sem_wait`) operation. It is basically like acquiring a lock. The only difference here is that we have multiple states, which are captured in the multiple values that the variable `count` can take. In fact, a lock can be thought of as a binary semaphore – `count` is equal to 1.

Listing 5.8: The `sem_post` operation

```
/* execute atomically */
if ((count == 0)  && process_waiting())
    wake_from_wait_queue();
else
    count ++;
```

The other important function in the case of semaphores is the analog of the unlock function, which is known as the `post` operation (`sem_post`). It is also referred to as the *signal* operation. The code for `sem_post` or signaling the semaphore is shown in Listing 5.8. In this function, if `count` is equal to 0, and we are trying to post to the semaphore (`sem_post`), the kernel picks one process from the waiting queue of processes and activates it. The assumption here is that the wait queue is not empty. The process that was woken up is assumed to acquire the semaphore, but `count` still remains 0. If the wait queue is empty, then `count` is simply incremented. The same increment operation is done when `count` is non-zero. This basically means that a process is releasing the semaphore, which makes it more available. This fact is recorded in the incremented `count` variable.

We shall subsequently see that semaphores allow us to implement bounded queues very easily.

5.1.9 Condition Variables

Listing 5.9: Condition variables in pthreads

```
/* Define the lock and a condition variable */
pthread_mutex_t mlock;
pthread_cond_t cond;
pthread_cond_init (&count, NULL);

/* wait on the condition variable*/
pthread_mutex_lock (&mlock);
pthread_cond_wait (&cond, &mlock);
```

```
pthread_mutex_unlock (&mlock);

/* signal the condition variable */
pthread_mutex_lock (&mlock);
pthread_cond_signal (&cond);
pthread_mutex_unlock (&mlock);
```

Semaphores require OS support. An OS routine is needed to make a process wait in the wait queue and then wake a process up once there is a *post* operation on the semaphore. Pthreads provide another solution in user space that are known as condition variables.

Refer to Listing 5.9. We define a mutex lock `mlock` and a condition variable `cond`. To wait on a condition (similar to `sem_wait`), we need to first acquire the mutex lock `mlock`. This is because a lock is required to update the state associated with the condition. Note that this state needs to be updated within a critical section. This critical section is protected by `mlock`. The `pthread_cond_wait` function is used to wait on a condition variable. Note that this function takes two inputs: the condition variable `cond` and the lock associated with it `mlock`.

Another thread can signal the condition variable (similar to `sem_post`). This needs to wake up one of the waiting threads. Again, we acquire the lock first. The `pthread_cond_signal` function is used to signal the condition variable. A waiting process immediately wakes up, if there is one.

If we wish to wake up all the waiting threads, then the `pthread_cond_broadcast` function can be used.

Point 5.1.5

A condition variable is not a semaphore. A semaphore has a notion of memory – it stores a count. The count can be incremented even if there is no waiting thread. However, in the case of a condition variable, there is a much stronger coupling. Whenever a pthread signal or broadcast call is made, the threads that are waiting on the condition variable *at that exact point of time* are woken up. Condition variables do not per se have a notion of memory. They don't maintain any counts. They simply act as a rendezvous mechanism (meeting point) between signaling and waiting threads. Hence, in this case, it is possible that a signal may be made but at that point of time there is no waiting thread, and thus the signal will be lost. This is known as the *lost wakeup problem*.

5.1.10 Reader-Writer Lock

Till now, we have not differentiated between read operations that do not change the memory state and write operations that change the memory state. There is a need to differentiate between them in some cases, if we need greater efficiency.

Clearly, a reader and writer cannot operate concurrently at the same point of time without synchronization because of the possibility of data races. We thus envision two specialized locks as a part of the locking mechanism: a *read lock* and a *write lock*. The read lock allows multiple readers to operate in parallel on a concurrent object. This leads to high-performance implementations by enhancing read parallelism. Given that there are no concurrent writers, there

is no possibility of a data race. We also need an exclusive write lock that allows only one writer to execute operations on the object. No other reader or writer are allowed to work on the queue concurrently. It just allows one writer to change the state of the queue. This is a reader-writer lock: either allow multiple concurrent readers or a single writer.

Listing 5.10: Code of the reader-writer lock

```
void get_write_lock(){
    LOCK(\_\_rwlock);
}
void release_write_lock(){
    UNLOCK(\_\_rwlock);
}

void get_read_lock(){
    LOCK(\_\_rdlock);
    if (readers == 0) LOCK(\_\_rwlock);

    readers++;
    UNLOCK(\_\_rdlock);
}

void release_read_lock(){
    LOCK(\_\_rdlock);
    readers--;
    if (readers == 0)
        UNLOCK (\_\_rwlock);
    UNLOCK (\_\_rdlock);
}
```

The code for the reader-writer lock is shown in Listing 5.10. We are assuming two macros LOCK and UNLOCK. They take a lock (mutex) as their argument, and invoke the methods lock and unlock, respectively. We use two instances of locks: __rwlock (for both readers and writers) and __rdlock (only for readers). The prefix __ signifies that these are internal locks within the high-level reader-writer lock. These locks are meant for implementing the logic of the reader-writer lock.

Let's first look at the code of a writer. There are two methods that it can invoke: get_write_lock and release_write_lock. In this case, we need a global lock that needs to stop other readers and writers from progressing. This is why in the function get_write_lock, we wait on the lock __rwlock. If it is acquired, it means that no other process is active in the critical section. For releasing a write lock (release_write_lock), we just need to unlock __rwlock.

The read lock, on the other hand, is slightly more complicated. Refer to the function get_read_lock in Listing 5.10. We use another mutex lock called __rdlock. A reader waits to acquire it. The idea is to maintain a count of the number of readers. Since there are concurrent updates to the readers variable, it needs to be protected by the __rdlock mutex. After acquiring __rdlock, it is possible that the lock acquiring process may find that a writer is active. We need to explicitly check for this by checking if the number of readers, readers, is equal to 0 or not. If it is equal to 0, then it means that other readers are not

active – a writer could be active. Otherwise, it means that other readers are active, and a writer cannot be active.

If `readers = 0` we need to acquire `__rwlock` to stop writers or wait for the currently active writer to complete. The rest of the method is reasonably straightforward. We increment the number of readers and finally release `__rdlock` such that other readers can proceed.

Releasing the read lock is also simple. We subtract 1 from the number of readers after acquiring `__rdlock`. If the number of readers becomes equal to 0, then there is no reason to hold the global `__rwlock`. It needs to be released such that writers can potentially get a chance to complete their operation.

A discerning reader at this point of time will clearly see that if readers are active, then new readers can keep coming in and the waiting write operation will never get a chance. This means that there is a possibility of starvation. Because `readers` may never reach 0, `__rwlock` will never be released by the reader holding it. The locks themselves could be fair, but overall we cannot guarantee fairness for writes. Hence, this version of the reader-writer lock's design needs improvement. Starvation-freedom is needed, especially for write operations. Various solutions to this problem are proposed in reference [Herlihy and Shavit, 2012].
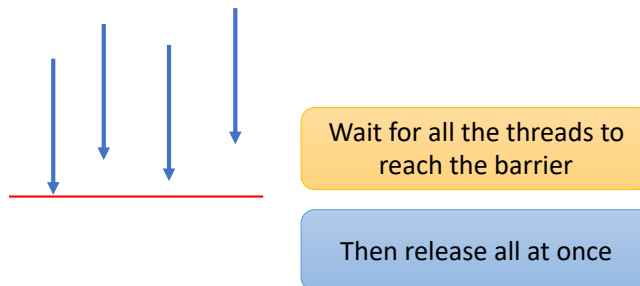
5.1.11 Barriers and Phasers
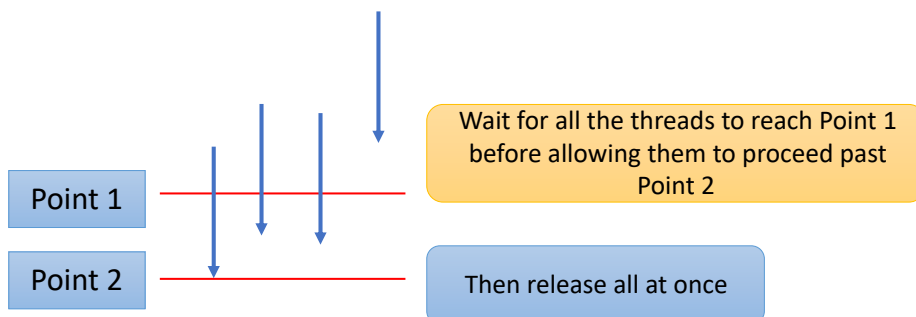

Figure 5.13: Barriers

Figure 5.14: Phasers

Let us now discuss two more important synchronization primitives: barriers and phasers. In general, in a parallel program, there is a need for a *rendezvous point*. We want all the threads to reach this rendezvous point before any thread is allowed to proceed beyond it. For example, in any map-reduce kind of computation, we typically require such rendezvous points. Let's say that we would like to add all the elements in a large array in parallel.

We can split the array into n chunks, where n is the number of threads. The strategy is to assign the ith chunk to the ith thread (map phase). Each thread can then add all the elements in its respective chunk, and then send the computed partial sum to a pre-designated root thread. The root thread needs to wait for all the threads to finish so that it can collect all the partial sums and add them to produce the final result (reduce phase). This is a rendezvous point insofar as all the threads are concerned because all of them need to reach this point before they can proceed to do other work. Such a point arises very commonly in a lot of scientific kernels that involve linear algebra.

Hence, it is very important to quickly realize such operations. Such operations are known as *barriers*. Note that this barrier is different from a memory barrier (discussed earlier), which is a fence operation. They just happen to share the same name (unfortunately so). We can psychologically think of a barrier as a point that stops threads from progressing, unless all the threads that are a part of the thread group associated with the barrier reach it (see Figure 5.13). Almost all programming languages, especially parallel programming languages provide support for barriers. In fact, supercomputers have special dedicated hardware for barrier operations. They can be realized very quickly, often in less than a few milliseconds.

There is a more flexible version of a barrier known as a phaser (see Figure 5.14). It is somewhat uncommon, but many languages such as Java define them and in many cases they prove to be very useful. In this case, we define two points in the code: Point 1 and Point 2. The rule is that no thread can cross Point 2 unless all the threads have arrived at Point 1. Point 1 is a point in the program, which in a certain sense precedes Point 2 or is before Point 2 in program order. Often when we are pipelining computations, there is a need for using phasers. We want some amount of work to be completed before some new work can be assigned to all the threads. Essentially, we want all the threads to complete the phase prior to Point 1, and enter the phase between Points 1 and 2, before a thread is allowed to enter the phase that succeeds Point 2.

5.2 Queues

Let us now see how to use all the synchronization primitives introduced in Section 5.1.

One of the most important data structures in a complex software system such as an OS kernel is a *queue*. All practical queues have a *bounded size*. Hence, we shall only refer to fixed-sized queues in the subsequent discussion. Typically, to communicate messages between different subsystems, queues are used as opposed to direct function calls or writing entries to an array. Queues provide the FIFO property, which also enforces an implicit notion of priority. They naturally enable asynchronous interaction. The producer can just enqueue an item in a queue and leave. The consumer can collect it much later. In a mul-

tithreaded kernel, there are typically many producers and consumers. Hence, a concurrent queue is a very important data structure in the kernel.

We can opt for a lock-free linearizable implementation, or use a version with locks. The choice of the data structure depends on the degree of contention, number of threads in the system and the desired simplicity of the implementation.

Producers   Consumers

Figure 5.15: A bounded queue

A conceptual view of a concurrent queue is shown in Figure 5.15, where we can observe multiple producers and consumers.

Let us start with outlining the conventions that we shall use in this section. A bounded queue is implemented as a circular buffer. We use an array with `BUFSIZE` entries and two pointers: `head` and `tail`. Entries are enqueued on the `tail` side, whereas they are dequeued on the `head` side. After enqueuing or dequeuing, we simply increment the `head` and `tail` pointers. This is an increment with a wraparound (modulo `BUFSIZE`). We use the macro `INC(x)`, which is implemented as `(x+1)%BUFSIZE`. This modulo addition provides the illusion of a circular buffer.

The convention we shall use is if `tail` = `head`, it means that the queue is *empty*. Otherwise, if there are entries, we simply dequeue the current head. We know that the queue is *full*, when we cannot add any other entry. This means that `INC(tail)==head`. We cannot increment tail, because that would make `tail == head`, which would also mean that the queue is empty. Hence, we stop when the "queue full" condition has been reached. If the queue is not full, then we add the new entry at the position of the current tail, and increment the tail pointer. Note that the tail pointer does not point to the last entry in the queue. It points to the first free entry in the buffer after the last entry in the queue. This means that if a new entry is to be enqueued, it is added at the `tail` position.

Finally, note that shared variables such as the head and tail pointers, and the array, are typically declared as `volatile` variables in C and C++. They are then not stored in registers but in the caches. Owing to cache coherence, changes made on one core are quickly visible on other cores.

Point 5.2.1

We always dequeue the current head, and we always enqueue at the current tail. The corresponding pointer is subsequently incremented (module BUFSIZE). If `tail` = `head`, the queue is empty. If `INC(tail)` = `head`, the queue is full. It is important to note that in this scheme, we do not use all the entries in the array to store queue elements. At least one element is kept empty. This helps us differentiate between queue empty and full conditions.

5.2.1 Wait-Free Queue

Listing 5.11 shows a queue that allows just one enqueuing thread and one dequeuing thread. No other threads are allowed to use this queue and also a thread cannot change its role. This means that the enqueuing thread cannot dequeue and vice versa. We use the code in Listing 5.11 as a running example. Most of the functions will remain the same across all the queue implementations.

Using this restriction, it turns out that we can easily create a wait-free queue. There is no need to use any locks, and operations complete within bounded time.

Listing 5.11: A simple wait-free queue with one enqueuer and one dequeuer

```c
#define BUFSIZE 10
#define INC(x) ((x+1)%BUFSIZE)
#define NUM 25

pthread_t tid[2];  /* has to be 2 here */
atomic_int queue[BUFSIZE];
atomic_int head=0, tail=0;

void nap(){
    struct timespec rem;
    int ms = rand() % 100;
    struct timespec req = {0, ms * 1000 * 1000};
    nanosleep(&req, &rem);
}

int enq (int val) {
    int cur_head = atomic_load (&head);
    int cur_tail = atomic_load (&tail);
    int new_tail = INC(cur_tail);

    /* check if the queue is full */
    if (new_tail == cur_head)
        return -1;

    /* There are no other enqueuers */
    atomic_store (&queue[cur_tail],val);
    atomic_store (&tail, new_tail);

    return 0; /* success */
}

int deq () {
    int cur_head = atomic_load (&head);
    int cur_tail = atomic_load (&tail);
    int new_head = INC(cur_head);

    /* check if the queue is empty*/
    if (cur_tail == cur_head)
        return -1;

    /* There are no other dequeuers */
    int val = atomic_load (&queue[cur_head]);
    atomic_store (&head, new_head);
```

```c
}

void* enqfunc (void *arg) {
    int i, val;
    int thread = *((int *) arg);
    srand(thread);

    for (i=0; i< NUM; i++){
        val = rand()%10;
        enq (val);
        nap();
    }
}

void* deqfunc (void *arg){
    int i, val;
    int thread = *((int *) arg);
    srand(thread);

    for (i=0; i< NUM; i++){
        val = deq();
        nap();
    }
}
int main() {
    int errcode, i = 0; int *ptr;
    void* (*fptr) (void*);

    for (i=0; i < 2; i++)
    {
        ptr = (int *) malloc (sizeof(int));
        *ptr = i;
        fptr = (i%2)? &enqfunc : &deqfunc;

        errcode = pthread_create(&(tid[i]), NULL,
                fptr, ptr);
    if (errcode)
        printf("Error in creating pthreads \n");
    }

    pthread_join (tid[0], NULL);
    pthread_join (tid[1], NULL);
}
```

The `main` function creates two threads. The odd-numbered thread enqueues by calling `enqfunc`, and the even-numbered thread dequeues by calling `deqfunc`. These functions invoke the `enq` and `deq` functions NUM times, respectively. Between iterations, the threads take a nap for a random duration.

The exact proof of wait freedom can be found in textbooks on this topic such as the book by Herlihy and Shavit [Herlihy and Shavit, 2012]. It is easy to see why this is the case. Given that there are no loops, we don't have a possibility of looping endlessly. Hence, the enqueue and dequeue operations will complete in bounded time. The proof of linearizability and correctness needs

more understanding and thus is beyond the scope of this book.

Note the use of *atomics* in the code. They are a staple of modern programming languages such as C++ 20 and other recent languages. Along with atomic load and store operations, the library provides many more functions such as `atomic_fetch_add`, `atomic_flag_test_and_set` and `atomic_compare_exchange_strong`. Depending upon the architecture and the function arguments, their implementations come with different memory ordering guarantees (embed different kinds of fences).

Other than the deliberate use of atomic variables and atomic read/write functions, this piece of code is very similar to the implementation of a bounded queue in a purely sequential system. There is no other major difference. The logic is the same. A couple of points need to be made. The first is note that in the `enq` function, we store the new entry first and then increment the `tail` pointer. Similarly, in the `deq` function, we read the value stored at the head of the queue first and then increment the `head` pointer. This is a standard pattern in programming concurrent systems. The values are first read or updated. Just reading the queue or writing to an empty entry does not change the state of the queue globally. The changes are not visible to other threads because they cannot see reads and access entries that are not between the `head` and `tail` pointers. Hence, the first action can be considered to be a local action that has no global visibility. After it has been executed, the state of the queue needs to be changed, which is realized by modifying the values of the `head` and `tail` pointers. Now the changes are globally visible. Other threads can see the results of the enqueue and dequeue operations. Given that we are assuming that the delay between consecutive instructions can be indefinite, it is always advisable to do the local changes first and do the global changes at the end. Typically, the global changes are the points of linearizability and thus all the changes to the data structure should have been made before them. The role of the globally visible actions is just to make those changes globally visible.

5.2.2 Queue with Mutexes

Let us now use the same basic template (in Listing 5.11) and create a generic version that allows any number of concurrent enqueuers and dequeuers. It is a blocking implementation. An enqueuer waits till there is at least one free entry in the queue. Similarly, a dequeuer waits till there is at least one entry in the queue. We shall opt for a version that uses mutexes (locks). Linux pthreads use *futexes* that are advanced versions of mutexes, where threads first try to acquire the lock using busy waiting and atomic instructions. If they are unsuccessful, then after some time, they request the operating system to swap them out such that other threads get a chance to execute. After all, spinlocks are a waste of time, and thus it is a much better idea to let other threads execute including the thread that currently holds the lock.

Listing 5.12: A queue with mutexes

```
#define LOCK(x) (pthread_mutex_lock(& x))
#define UNLOCK(x) (pthread_mutex_unlock(& x))
pthread_mutex_t qlock;

int enq (int val) {
```

```
    int status;
    do {
        LOCK(qlock);
        if (INC(tail) == head) status = -1; /* full */
        else {
            queue[tail] = val;
            tail = INC(tail);
            status = 0;
        }
        UNLOCK(qlock);
    } while (status == -1);
    return status;
}

int deq () {
    int val, status;

    do {
        LOCK (qlock);
        if (tail == head) status = -1;  /* empty */
        else {
            val = queue[head];
            head = INC(head);
            status = 0;
        }
        UNLOCK (qlock);
    } while (status == -1);
    return val;
}
int main() {
    ...
    pthread_mutex_init (&qlock, NULL);
    ...
    pthread_mutex_destroy (&qlock);
}
```

We define a pthread mutex `qlock`. It needs to be initialized using the `pthread_mutex_init` call. The first argument is a pointer to the lock and the second argument is a pointer to a pthread attributes structure (specifies the behavior of the lock). In this case it is NULL because there are no additional attributes. In the `main` function, after all the processing is done, the lock is ultimately freed (destroyed).

We define two macros `LOCK` and `UNLOCK` that wrap the pthread functions `pthread_mutex_lock` and `pthread_mutex_unlock`, respectively.

The code in the `enq` and `deq` functions is straightforward – it is just protected by a lock. The code keeps looping until an entry is successfully enqueued or dequeued.

5.2.3   Queue with Semaphores

Let us now implement a bounded queue with semaphores. The additional/modified code is shown in Listing 5.13.

Listing 5.13: A queue with semaphores

```c
#define LOCK(x) (sem_wait(& x))
#define UNLOCK(x) (sem_post(& x))
sem_t qlock;

...

int main() {
    sem_init (&qlock, 0, 1);
    ...
    sem_destroy(&qlock);
}
```

We initialize a semaphore using the sem_init call. It takes as arguments a pointer to the semaphore, whether it is shared between processes (1) or just shared between different threads of a multithreaded process (0), and the initial value of the count (1 in this case). Finally, the semaphore needs to be destroyed using the call sem_destroy.

We redefine the LOCK and UNLOCK macros, using the sem_wait and sem_post calls, respectively. The rest of the code remains the same. Here, we are just using semaphores as locks (binary semaphores). The code uses busy waiting, which as we have argued is not desirable. We are not using the full power of semaphores.

It is easy to see why a sem_wait call is equivalent to a lock operation. If the lock is free, then the value of the count in the semaphore is 1. It is simply decremented and set to 0. This means that the lock has been acquired. However, if the count was 0, then there is a need to wait for it to become 1.

Similarly, a sem_post call is equivalent to an unlock. If there is a waiting process/thread, it is woken up. It is provided access to the critical section. This is like an unlock operation followed by a lock operation. However, if there is no waiting process, then the count is incremented. This means that the lock is freed. Subsequently, the equivalent lock can be locked.

Someone may want to argue that it is possible to have a count that is more than 1. This is indeed possible if there are many back-to-back sem_post calls. However, the expectation is that the programmers will be disciplined and they will not have patterns of this kind. Every call to the LOCK macro will be matched by a call to the UNLOCK macro and vice versa. Hence, there is no possibility of this happening.

Next, let us use the real power of semaphores. They are ideal for implementing bounded queues.

5.2.4 Queue with Semaphores but No Busy Waiting

Listing 5.14 shows the code of one such queue that uses semaphores but does not have busy waiting.

Listing 5.14: A queue with semaphores but does not have busy waiting

```c
#define WAIT(x) (sem_wait(& x))
#define POST(x) (sem_post(& x))
sem_t qlock, empty, full;
```

```c
int enq (int val) {
    WAIT(empty);
    WAIT(qlock);

    queue[tail] = val;
    tail = INC(tail);

    POST(qlock);
    POST(full);

    return 0; /* success */
}

int deq () {
    WAIT(full);
    WAIT(qlock);

    int val = queue[head];
    head = INC(head);

    POST(qlock);
    POST(empty);

    return val;
}

int main() {
    sem_init (&qlock, 0, 1);
    sem_init (&empty, 0, BUFSIZE);
    sem_init (&full, 0, 0);
    ...
    sem_destroy(&qlock);
    sem_destroy(&empty);
    sem_destroy(&full);
}
```

We use three semaphores here. We still use `qlock`, which is needed to protect the shared variables. Additionally, we use the semaphore `empty` that is initialized to `BUFSIZE` (maximum size of the queue) and the `full` semaphore that is initialized to 0. These will be used for waking up threads that are waiting. We define the `WAIT` and `POST` macros that wrap `sem_wait` and `sem_post`, respectively.

Consider the `enq` function. We first wait on the `empty` semaphore. There need to be free entries available. Initially, we have `BUFSIZE` free entries. Every time a thread calls `sem_wait` on the semaphore, it decrements the number of free entries by 1 until the count reaches 0. After that the thread waits. Once it is released, we can be sure that there is at least one free slot in the queue. Otherwise the `sem_wait` call would not have been successful. It was successful because it was able to decrement the non-zero count. This can only happen if there was at least one free entry. We can think of this entry getting reserved for the current thread (in an implicit sense).

Subsequently, we enter the critical section that is protected by the binary semaphore `qlock`. There is no need to perform any check on whether the queue

is full or not. We know that it is not full because the thread successfully acquired the `empty` semaphore. After releasing `qlock`, we signal the `full` semaphore. This indicates that an entry has been added to the queue. In terms of semantics, it has the reverse connotation as `empty`.

Let us now look at the `deq` function. It follows the reverse logic. We start out by waiting on the `full` semaphore. There needs to at least be one entry in the queue. This means that the count associated with `full` should be non-zero. Once this semaphore has been acquired, we are sure that there is at least one entry in the queue, and it will remain there until it is dequeued (property of the semaphore). The critical section again need not have any checks regarding whether the queue is empty or not. It needs to be protected by the `qlock` binary semaphore to make the program data-race-free. Finally, we complete the function by signaling the `empty` semaphore. The reason for this is that we are removing an entry from the queue, or creating one additional free entry. Waiting enqueuers will get signaled.

Note that there is no busy waiting. Threads either immediately acquire the semaphore if the count is non-zero or are swapped out. They are put in a wait queue inside the kernel. They thus do not monopolize CPU resources and more useful work is done. We are also utilizing the natural strength of semaphores.

5.2.5 Reader-Writer Lock

Let us now add a new function in our queue, which is a `peak` function. It allows us to read the value at the head of the queue without actually removing it. This function turns out to be quite useful in many scenarios. It is very different in character. As compared to the regular enqueue and dequeue functions, the `peak` function is a read-only method that does not change the state of the queue. Enqueue and dequeue operations, which actually change the state of the queue, are akin to writes. It is thus a fit case for using a *reader-writer lock*.

In the `peak` function, we need to do the following. If `head` is equal to `tail`, then we return -1 (the queue is empty). Otherwise, we return the contents of the head of the queue. Note that read operations do not interfere with each other; hence, they can execute concurrently (such as the `peak` function).

However, we cannot allow a parallel enqueue or dequeue – they are essentially write operations. There will be a data race condition here, and thus some form of synchronization will be required. Our aim is to allow multiple readers to read (peak) together, but only allow a single writer to change the state of the queue (enqueue or dequeue). Let us use the functions of the reader-writer lock to create such a queue (see Section 5.1.10).

Listing 5.15 shows the additional/modified code for a queue with a reader-writer lock. We reuse the code for the reader-writer lock that we had shown in Listing 5.10. The pthreads library does provide a reader-writer lock facility (`pthread_rwlock_t`) on some platforms, however, we prefer to use our own code.

The `peak` function uses the read lock. It acquires it using the `get_read_lock` function. That is all that is required for it to execute correctly. Multiple readers can execute concurrently. No writer can come in until there is a reader in the system.

Listing 5.15: A queue with reader-writer locks

```
sem_t rwlock, read_lock, full, empty;
```

```
int peak() {
    /* This is a read function */
    get_read_lock();
    int val = (head == tail)? -1 : queue[head];
    release_read_lock();

    return val;
}
int enq (int val) {
    WAIT(empty);

    /* Get the write lock and perform the enqueue*/
    get_write_lock();
    queue[tail] = val;
    tail = INC(tail);
    release_write_lock();

    POST(full);
    return 0; /* success */
}

int deq () {
    int val;

    WAIT(full);

    /* Get the write lock and perform the dequeue */
    get_write_lock();
    val = queue[head];
    head = INC(head);
    release_write_lock();

    POST(empty);
    return val;
}
```

The code of the `enq` and `deq` functions remain more or less the same. We wait and signal the same set of semaphores: `empty` and `full`. The only difference is that we do not acquire a generic lock, but we acquire the write lock using the `get_write_lock` function. This does not make a difference because there are no other concurrent readers or writers.

The only difference is just that we are using a different set of locks for the `peak` function and the `enq/deq` functions. There is a performance advantage because we allow multiple readers to do their work in parallel.

5.2.6 Linux Message Queues

Example 5.2.1

Use the Linux message queues to send a string from a sending process to a receiving process.

Answer:

Listing 5.16: Sending a message

```c
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>

#define MAX_TEXT 512

struct msg_buffer {
    long msg_type;
    char msg_text[MAX_TEXT];
};

int main() {
    // generate a unique key
    key_t key = ftok("msgfile", 65);

    // create the message queue
    int msgid = msgget(key, 0666 | IPC_CREAT);

    // populate the message buffer
    struct msg_buffer message;
    message.msg_type = 1;
    strcpy(message.msg_text, "Hello from sender!");

    // send the message
    msgsnd(msgid, &message, sizeof(message.msg_text),
        0);

    printf("Sent: %s\n", message.msg_text);

    return 0;
}
```

Listing 5.17: Receiving a message

```c
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_TEXT 512

struct msg_buffer {
    long msg_type;
```

```
    char msg_text[MAX_TEXT];
};

int main() {
    // Use the same key as the sender
    key_t key = ftok("msgfile", 65);

    // get the message queue id
    int msgid = msgget(key, 0666 | IPC_CREAT);

    // receive a message of type 1
    struct msg_buffer message;
    msgrcv(msgid, &message, sizeof(message.msg_text),
        1, 0);

    printf("Received: %s\n", message.msg_text);

    // destroy the message queue
    msgctl(msgid, IPC_RMID, NULL);

    return 0;
}
```

To use any Linux kernel IPC (inter-process communication) facility, it is necessary to first create a unique key and then associate it with a message queue. This is achieved in the first two statements of the `main` function of the sending process.

Linux internally implements a queue of `msg_buffer` structures. Eech such structure has two arguments: a message type (`long`) and an array of characters(`msg_text`). The `msgsnd` function sends a pointer to this message buffer along with the size of the string contained in it to the message queue. This is all that needs to be done to send a message.

The code of the message receiver is quite similar. Here also we obtain a key and find the id of the corresponding message queue. Next, we use the `msgrcv` function call. We pass a pointer to the message buffer and the type of the message. The message queue logic in the kernel fills up the message buffer with the contents of the earliest message that matches the type. Note that we are passing a pointer to a stack variable (`message`), which is in user space. Finally, the message queue can be destroyed by calling the `msgctl` function.

5.3 Concurrency within the Kernel

Let us now look at concurrency within the kernel. As we have discussed earlier, we typically refer to kernel processes as *kernel threads* because they share their address space with each other. Hence, concurrency per se is a very important issue in the kernel code. Ensuring correctness, especially freedom from deadlocks, livelocks, starvation and data races is of utmost importance.

Linux internally refers to a multicore processor as a symmetric multiprocessor (smp). The computing units typically have equal/similar access to memory

and I/O devices. However, this is not strictly necessary. There can be NUMA (non-uniform memory access) machines where the memory access time is not constant. Different cores have different memory access latencies, and the latency depends on the memory module being accessed.

5.3.1 Kernel-Level Locking: Spinlocks

We have two options: we can either use regular code with locks or we can use lock-free data structures. As we have argued earlier, lock-free variants of data structures are sometimes very useful. They are often high-performance implementations owing to the fact that they do not use locks. It is not possible to have deadlocks with lock-free algorithms. Even if a thread goes off to sleep or is swapped out, there is no problem. The only shortcoming of lock-free code is that it can sometimes lead to starvation. This is very rare in practice though. We can always increase the priority of the thread that is supposedly starving. On the flip side, for a large number of data structures, writing correct and efficient lock-free code is very difficult, and writing wait-free code is even more difficult. They are hard to write, verify and debug. Hence, a large part of the kernel still uses regular spinlocks, which are busy-waiting locks. However, they come with a twist.

They observe a few additional restrictions. Unlike regular mutexes that are used in user space, the thread holding the spinlock is not allowed to go to sleep, get migrated or get swapped out (preempted). This means that interrupts need to be disabled in the critical section (protected by kernel spinlocks). This further implies that these locks can also be used in the interrupt context (also known as the atomic context in the kernel). A thread holding such a lock will complete in a finite amount of time unless it is a part of a deadlock (discussed later). On a multicore machine, it is possible that a thread may wait for the lock to be released by a thread running on another core. Given that the lock holder cannot block or sleep, this mechanism is effectively equivalent to a lock-free algorithm in terms of performance. Needless to say, we are assuming that the lock holder will complete the critical section in a finite amount of time and not rely on operations such as I/O accesses that can take an indefinite amount of time or require the thread to sleep. This will indeed be the case given our restrictions on blocking interrupts and disallowing preemption.

If we were to allow context switching after a spinlock has been acquired, then we may have a deadlock situation The new thread may have a higher priority. To make matters worse, it may try to acquire the spinlock. Given that we have busy waiting, it will continue to loop and wait for the lock to get freed. But the lock may never get freed because the thread that is holding the lock may never get a chance to run. The reason it may not get a chance to run is because it has a lower priority than the thread that is waiting on the lock. Hence, kernel-level spinlocks need these restrictions. A spinlock effectively locks the CPU. The lock-holding thread does not migrate, nor does it allow any other thread to run until it has finished executing the critical section and released the spinlock.

> **Point 5.3.1**
>
> A spinlock is effectively a lock on the CPU because no other thread is allowed to run on the CPU till it is released.

Enabling and Disabling Preemption

Enabling and disabling preemption is an operation that needs to be done very frequently. Given the fact that it is now associated with spinlocks, which we expect to use frequently in kernel code, efficiency is paramount. The expectation is that acquiring and releasing a spinlock should be a very fast operation. Hence, enabling and disabling preemption on a core should also be a very fast operation. This is indeed the case (refer to Listing 5.18). There is a macro called `preempt_disable`, which uses a logic similar to semaphores.

Listing 5.18: Code to enable and disable preemption
source : include/linux/preempt.h#L201

```
#define preempt_disable() \
do { \
    preempt_count_inc(); \
    barrier(); \
} while (0)

#define preempt_enable() \
do { \
    barrier(); \
    if (unlikely(preempt_count_dec_and_test())) \
        __preempt_schedule(); \
} while (0)
```

The key idea is as follows. A preemption count variable is maintained. If the count is non-zero, then it means that preemption is not allowed. Whereas if the count is 0, it means that preemption is allowed. If we want to disable preemption, all that we have to do is increment the count and also insert a fence operation, which is also known as a memory barrier. The reason for adding a memory barrier is to ensure that the code in the critical section is not reordered and brought before the write operation to the preemption count variable. Note that this is not the same barrier that we discussed in the section on barriers and phasers (Section 5.1.11). They just happen to share the same name. They are synchronization operations, whereas the memory barrier is a fence, which basically disables memory reordering. The preemption count is stored in a per-CPU region of memory (accessible via the `gs` segment register). Accessing it is a very fast operation and requires very few instructions (as we have seen before in the case of the `current` pointer).

The code for enabling preemption is shown in Listing 5.18. In this case, we do more or less the reverse. We have a fence operation to ensure that all the pending memory operations (executed in the critical section) completely finish and are visible to all the threads. After that, we decrement the preemption count using an atomic operation. If the count reaches zero, it means that preemption is allowed. It is necessary to call the `schedule` function to select the next task that needs to execute on the core. An astute reader will figure out that this operation is like a semaphore, where if preemption is disabled n times, it needs to be enabled n times for the task running on the core to become preemptible.

> **Trivia 5.3.1**
>
> Assume that a task acquires n spinlocks one after the other. This means that preemption is disabled n times. This fact needs to be recorded in the value of the preemption count by incrementing it n times. Preemption can only be enabled when all the spinlocks are released. Each spinlock release operation decrements the preemption count. After decrementing the count variable n times, it reverts to the value it used to have before the task acquired the n locks. At this point of time if preemption was allowed, then after acquiring and releasing all n locks, preemption will be allowed again.

Spinlock: Kernel Code

Listing 5.19: Wrapper of a spinlock
source : include/linux/spinlock_types_raw.h#L14

```
typedef struct raw_spinlock {
    arch_spinlock_t raw_lock;
    #ifdef CONFIG_DEBUG_LOCK_ALLOC
        struct lockdep_map dep_map;
    #endif
} raw_spinlock_t;
```

The code for a spinlock is shown in Listing 5.19. We see that the spinlock structure encapsulates an `arch_spinlock_t` lock and a dependency map (`struct lockdep_map`). The `raw_lock` member is the actual spinlock. The dependency map is used to check for deadlocks (we shall discuss this later).

Listing 5.20: Inner workings of a spinlock
source : include/asm − generic/spinlock.h#L33

```
void arch_spin_lock(arch_spinlock_t *lock) {
    u32 val = atomic_fetch_add (1<<16, lock);
    u16 ticket = val >> 16;   /* upper 16 bits of lock */
    if (ticket == (u16) val)  /* Ticket id == ticket next in
        line */
        return;
    atomic_cond_read_acquire(lock, ticket == (u16)VAL);
    smp_mb();  /* barrier instruction*/
}
```

Let us understand the design of the spinlock. Its code is shown in Listing 5.20. It is a classic ticket lock that has two components: a *ticket*, which acts like a coupon, and the id of the next ticket that needs to be serviced (*next*) (to-be-serviced) (refer to Figure 5.16). Every time a thread tries to acquire a lock, it gets a new ticket. It is deemed to have acquired the lock when *ticket == next*.

Consider a typical bank where we go to meet a teller. We first get a coupon, which in this case is the ticket. Then we wait for our coupon/ticket number to be displayed. Once this happens, we can go to the counter at which a teller is waiting for us. The idea here is quite similar.
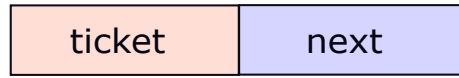
Figure 5.16: The lock variable with the *ticket* and *next* fields

If we think about it, we can easily conclude that this lock guarantees fairness – starvation is not possible. The way that this lock is designed in practice is quite interesting. Instead of using multiple fields, a single 32-bit unsigned integer is used to store or rather *pack* both the *ticket* and the *next* fields. This is the contents of the lock variable (pointed to by the `lock` pointer). We divide this unsigned 32-bit integer into two smaller unsigned integers that are each 16 bits wide. The upper 16 bits store the ticket id (*ticket*). The lower 16 bits store the value of the *next* field.

When a thread arrives, it tries to get a ticket. It uses the atomic fetch-and-add instruction to fetch the *ticket* and *next* fields atomically from the lock variable. These two fields are stored in the internal variable `val`. In the fetch-and-add atomic operation, the *ticket* part of the lock variable is also incremented. This is achieved by adding 2^{16} (1 ¡¡ 16) to the lock variable. Given that the *ticket* is field is offset by 16 bits, this operation has the effect of incrementing the value of *ticket* in the lock variable. Note that this instruction has a built-in memory fence as well (more about this later). Now, the original ticket can be extracted quite easily by right shifting the value (`val`) returned by the fetch-and-add instruction by 16 positions. This is done in the next line.

The next task is to extract the lower 16 bits (*next* field). This is the number of the ticket that is the holder of the lock, which basically means that if the current ticket is equal to these 16 bits, then we can go ahead and execute the critical section. This is easy to do using a simple typecast operation. Here, the type `u16` refers to a 16-bit unsigned integer. Simply typecasting `val` to the type `u16` has the effect of retrieving the lower 16 bits as an unsigned integer. This is all that we need to do. Next, we compare this value with the thread's ticket, which is also a 16-bit unsigned integer. If both are equal, then the spinlock has effectively been acquired and the method can return.

Now, assume that they are not equal, which is the more common case. Then there is a need to perform busy-waiting. This is where we call the macro `atomic_cond_read_acquire`, which requires two arguments: the lock value and the condition that needs to be true. Note the unusual semantics of a C macro that takes a variable and a condition as an argument. Up till now, we have not seen such a pattern. This condition checks whether the current value of the ticket in the lock variable is equal to the *next* field in the lock variable. It ends up calling the macro `smp_cond_load_relaxed`, which resolves to a macro whose code is shown in Listing 5.21.

Listing 5.21: The code for the busy-wait loop
source : include/asm − generic/barrier.h#L248

```
#define smp_cond_load_relaxed(ptr, cond_expr) ({    \
    typeof(ptr) __PTR = (ptr);                       \
    __unqual_scalar_typeof(*ptr) VAL;                \
    for (;;) {                                       \
        VAL = READ_ONCE(*__PTR);                     \
```

```
        if (cond_expr)                                 \
            break;                                     \
        cpu_relax();      /* insert a delay*/          \
    }                                                  \
    (typeof(*ptr)) VAL;                                \
})
```

The inputs are a pointer to the lock variable and an expression that needs to evaluate to true. Then we have an infinite loop where we dereference the pointer and fetch the current value of the lock. Next, we evaluate the conditional expression (`ticket == (u16)VAL`). If the conditional expression evaluates to true, then it means that the lock has been acquired. We can then break from the infinite loop and resume the rest of the execution. Note that we cannot return from a macro because a macro is just a piece of code that is copy-pasted by the preprocessor with appropriate argument substitutions.

In case the conditional expression evaluates to false, then of course, there is a need to keep iterating. But along with that, we would not like to contend for the lock all the time. This would lead to a lot of cache line bouncing across cores, which is detrimental to performance. We are unnecessarily increasing the memory and on-chip network traffic. It is a better idea to wait for some time and try again. This is where the function `cpu_relax` is used. It makes the thread back off for some time.

Note that fairness is not guaranteed. However, in all practical situations, we can expect that the lock will ultimately be acquired. Subsequently, there is a need to execute a memory barrier (fence). Note that this is a generic pattern? Whenever we acquire a lock, there is a need to insert a memory barrier after it such that the state of the lock variable is globally visible before the updates made in the critical section are visible. This ensures that changes made in the critical section get reflected only after the lock has been acquired.

Listing 5.22: The code for unlocking a spinlock
source : include/asm − generic/spinlock.h#L63

```
void arch_spin_unlock(arch_spinlock_t *lock)
{
    u16 *ptr = (u16 *)lock + IS_ENABLED(
        CONFIG_CPU_BIG_ENDIAN);
    u32 val = atomic_read(lock);
    smp_store_release(ptr, (u16)val + 1);    /* store
        following release consistency semantics */
}
```

Let us now come to the unlock function. This is shown in Listing 5.22. It is quite straightforward. The first task is to find the address of the *next* field. This needs to be incremented to let the new owner of the lock know that it can now proceed. There is a complication here. We need to see if the machine is big endian or little endian. If it is a big endian machine, which basically means that the lower 16 bits are actually stored in the higher addresses, then a small correction to the address needs to be made. This logic is embedded in the `IS_ENABLED` (big endian) macro. It returns 1 for a big endian machine, which means that the lock address is incremented by the size of a `u16` number (=2 bytes).

Regardless of the endianness, at the end of this statement, the address of the *next* field is stored in the `ptr` variable. Next, we read the lock variable and extract the *next* field. The last line increments it by 1 and stores the result in the address pointed to by `ptr`, Effectively, the *next* field in the lock gets incremented. Now, if there is a thread whose ticket number is equal to the contents of the *next* field, then it knows that it is the new owner of the lock. It can proceed with completing the process of lock acquisition and start executing the critical section.

Finally, note that the `smp_store_release` macro also includes a fence. This ensures that all the writes made in the critical section are visible to the rest of the threads after the lock has been released. This completes the unlock process.

Fast Path and Slow Path Approach

Listing 5.23: The code to try to acquire a spinlock (fast path)
source : include/asm − generic/spinlock.h#L53

```
static __always_inline bool arch_spin_trylock(
    arch_spinlock_t *lock)
{
    u32 old = atomic_read(lock);
    if ((old >> 16) != (old & 0xffff))
        return false;
    return atomic_try_cmpxchg(lock, &old, old + (1<<16));
}
```

The fast path and slow path approach is a standard mechanism to speed up the process of acquiring locks. In fact, it is a generic paradigm where there is a fast path that is used when there is minimal contention for a shared resource like a lock. In this path threads try to directly acquire the lock by modifying the lock variable. They do not perform busy-waiting. If they are not successful, which will happen if there is contention, then they revert to the slow path.

Listing 5.23 shows one such function in which we try to acquire the lock. If we are not successful, then we return false. This further means that the system automatically reverts to the slow path, which was shown in Listing 5.20.

In this case, we first read the value of the `lock` variable. Then we quickly compare the value of the *next* field (`old & 0xffff`) with the ticket (`old >> 16`). If they are not the same, then we can return from the function returning false. This basically means that we need to wait to acquire the lock. However, if the values are equal, then an attempt should be made to acquire the lock. This is where we attempt an atomic compare and exchange (last line). If the value of the `lock` variable has not changed, then we try to set it to (`old + (1 << 16)`). We are basically adding 1 to the upper 16 bits of the `lock` variable. This means that we are incrementing the ticket number by 1, which is something that we would have done anyway, had we followed the slow path (code in Listing 5.20). We try this fast path code only once, or maybe a few times, and if we are not successful in acquiring the lock, then there is a need to fall back to the regular slow path code (`arch_spin_lock`).

Bear in mind that is a generic mechanism, and it can be used for many other kinds of concurrent objects as well. The fast path captures the scenario

in which there is less contention and the slow path captures scenarios where the contention is from moderate to high.

5.3.2 Kernel Mutexes

A spinlock is held by the CPU (conceptually), however, the kernel mutex is held by a task. It is per se not tied to a CPU.

Listing 5.24: A kernel mutex
source : include/linux/mutex.h#L63

```
struct mutex {
    atomic_long_t       owner;
    raw_spinlock_t      wait_lock;
    struct list_head    wait_list;

#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map dep_map;
#endif
};
```

The code of the kernel mutex is shown in Listing 5.24. Along with a spinlock (`wait_lock`), it contains a pointer to the owner of the mutex and a waiting list of threads. Additionally, to prevent deadlocks it also has a pointer to a lock dependency map. However, this field is optional – it depends on the compilation parameters. Let us elaborate.

The `owner` field is a pointer to the `task_struct` of the owner. An astute reader may wonder why it is an `atomic_long_t` and not a `task_struct *`. Herein, lies a small and neat trick. We wish to provide a fast-path mechanism to acquire the lock. We would like the `owner` field to contain the value of the `task_struct` pointer of the lock-holding thread. However, we would also like to pack a few additional bits to indicate the status of the lock. They can indicate if the lock is acquired or not, if there are waiting threads, etc. We can thus perform a compare and swap (CAS) on the `owner` field to quickly get access to the lock. We try the fast path only once. This means that a thread compares the value stored in `owner` with 0 and then try to sets it to the `task_struct` of the current thread. Some additional bits are also set to indicate the status of the lock.

If the lock is currently acquired, we enter the slow path. In this case, the threads waiting to acquire the lock are stored in `wait_list`, which is protected by the spinlock `wait_lock`. This means that before enqueuing the current thread in `wait_list`, we need to acquire the spinlock `wait_lock` first.

Listing 5.25: The mutex lock operation
source : kernel/locking/mutex.c#L281

```
void mutex_lock(struct mutex *lock)
{
    might_sleep();  /* prints a stack trace if called in an
        atomic context (sleeping not allowed) */
    if (!__mutex_trylock_fast(lock))   /* cmpxchg on owner
        */
    __mutex_lock_slowpath(lock);
}
```

Listing 5.25 shows the code of the lock function (`mutex_lock`) in some more detail. Its only argument is a pointer to the mutex. First, there is a need to check if this call is being made in the right context or not. If the call is made in an *atomic* (also referred to as the *interrupt*) context in which the code cannot be preempted, sleeping and blocking are not allowed. Hence, if the mutex lock call has been made in this context, it is important to flag this event as an error and also print the stack trace (the function call path leading to the current function). Remedial action can be taken.

Assume that the check passes, and we are not in the atomic/interrupt context, then we first make an attempt to acquire the mutex via the fast path. If we are not successful, then we try to acquire the mutex via the slow path using the function `__mutex_lock_slowpath`.

The slow path is slightly tricky. Let us explain the main idea, then we will look at its nuances. Broadly speaking, we first try to acquire the spinlock, and if it is not possible then the kernel thread is added to the queue of waiting threads. Then, it goes to sleep. In general, the task is locked in the UNINTERRUPTIBLE state. This is because we don't want to wake it up to process signals. Kernel threads typically do not use signals. Hence, they have little to gain from being in an INTERRUPTIBLE state. Now, when the lock is released, the lock is handed over to a waiting thread if there is one.

Note that this is a kernel thread. Going to sleep does not mean going to sleep immediately. It just means setting the status of the task to either INTERRUPTIBLE or UNINTERRUPTIBLE. The task still runs. It needs to subsequently invoke the scheduler such that it can find the most eligible task to run on the core. Given the status of the current task is set to a sleep state, the scheduler will not choose it for execution. After it is swapped out, its sleep state begins.

The unlock process pretty much does the reverse. We first check if there are waiting tasks in the `wait_list`. If there are no waiting tasks, then the `owner` field can directly be set to 0, and we can return. However, if there are waiting tasks, then there is a need to hand over the lock to one of the waiting threads.

Race Conditions:
It is possible that there are race conditions. Let the lock-holding thread be T_l. Consider and thread T_w, which is trying to acquire the lock. Consider the following sequence of events.

1. T_w tries to acquire the lock in the fast path, it is not successful because T_l has acquired it.

2. Next, T_w prepares to enter the slow path and get queued in the list of waiting kernel tasks.

3. T_w gets delayed.

4. T_l releases the lock, checks the waiting queue of tasks, finds it to be empty and exits.

5. T_w wakes up and inserts itself in the queue.

6. There is no thread to wake it up. It remains in the queue forever.

This situation is indeed possible if we don't take additional care. Such race conditions are quite tricky.

Hence, we need to do much more processing. Let us look at the algorithm to acquire the lock in the slow path. Let us list the steps again keeping the race condition in mind.

1. Acquire `wait_lock` (the spinlock)

2. Try the fast path again. Attempt to acquire the lock using a CAS operation. If successful, release `wait_lock` and return.

3. Otherwise, insert the current task in `wait_list`.

4. Set the state to a sleep state (such as INTERRUPTIBLE).

5. Release `wait_lock`

We ensure that we attempt to take the lock again inside a critical section protected by `wait_lock`. If there is no success, then the task corresponding to the current thread is enqueued in `wait_list` and the spinlock is released. It is important to do a check again because it is possible that the lock is free now. It is true that the current thread had checked the lock variable earlier when it was executing its fast path. However, between both these events a lot of time could have elapsed.

Let us now consider the unlock algorithm. Here also, there is a need to be aware of such subtle race conditions.

1. Acquire `wait_lock`

2. Check if there is a waiting task in `wait_list`.

3. Let T be the first such task if there is one, NULL otherwise.

4. If `wait_list` is empty, set the lock variable (`owner` field) to 0 and return.

5. Otherwise, transfer the ownership of the mutex to T by setting the `owner` field to T.

6. Release `wait_lock`

7. Enable task T

Let us understand why race conditions are taken care of. Consider thread T_w that was not successful in acquiring the mutex in the first path. In this case, it enters the slow path. Note that it needs to check the lock variable (`owner`) again inside the critical section protected by `wait_lock`. If it is free, it will try to set it using a CAS operation; otherwise, it means that another thread T_a was successful. T_a is the new owner of the mutex. Even if it releases the mutex immediately, it cannot simply exit. It will have to acquire `wait_lock` and check the queue of the waiting tasks.

T_a cannot acquire `wait_lock` until T_w releases it. T_w will only release it when it has enqueued itself in `wait_list`. This means that when T_a acquires `wait_lock` and enters the critical section, it is bound to find a non-empty list of waiting tasks with T_w in it. At this point, T_w will be woken up. It is thus not possible for the owner of the mutex to miss T_w. Hence, this design is not vulnerable to such race conditions. If a thread has been enqueued, it is guaranteed to be woken up by some other thread.

Other Kinds of Locks

Note that the kernel code can use many other kinds of locks. Their code is available in the directory kernel/locking.

A notable example is a queue-based spin lock (MCS lock: `qspinlock` in kernel code). An MCS lock is in general known to be a very scalable lock that it is quite fast. It also minimizes cache line bouncing (movement of cache lines containing the lock variable across cores). The idea is that we create a linked list of nodes, where each node encapsulates a lock request. A lock request contains a pointer to the task that needs to acquire the lock and a lock variable. In addition to the linked list, there is a dedicated *tail* pointer that points to the end of the linked list. The end of the linked list is the most recently added node. The basic design is shown in Figure 5.17.
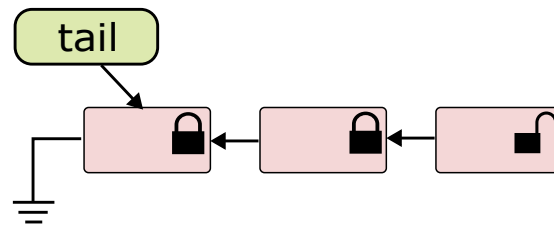
Figure 5.17: The MCS lock

The idea is to add the current node (wrapper of the `current` task) to the end of this list. Note that in this list, each node points to the node that was added right after it. A lock acquisition has two steps: make the current tail node point to the new node (containing the `current` task), and modify the *tail* pointer to point to the new node. This group of two operations needs to execute atomically – it needs to appear that both of the instructions executed together at a single point of time, instantaneously. The lock variable in each node indicates whether the lock is free or busy. Note that each node needs to perform busy waiting on a lock variable that is stored in its own node. This minimizes cache line bouncing.

When a node releases the MCS lock, it sets the lock variable of its successor to free. The task pointed to by the successor node can then acquire the lock.

The MCS lock is a very classical lock and almost all texts on concurrent systems discuss its design in great detail. Hence, we shall not delve further (reference [Herlihy and Shavit, 2012]).

There are a few more variants of locks supported by the kernel such as the `osq_lock` (variant of the MCS lock) and the `qrwlock` (reader-writer lock that gives priority to readers).

5.3.3   Kernel Semaphores

Listing 5.26: The kernel semaphore
source : include/linux/semaphore.h#L15

```
struct semaphore {
    raw_spinlock_t    lock;
```

```
    unsigned int    count;
    struct list_head  wait_list;
};
```

The kernel code has its version of semaphores (see Listing 5.26). It uses a spin lock (`lock`) to protect the semaphore variable `count`. Akin to user-level semaphores, the kernel semaphore supports two methods that correspond to wait and post, respectively. They are known as `down` (wait) and `up` (post/signal). The kernel semaphore functions in exactly the same manner as a user-level semaphore. After acquiring the lock, the `count` variable is decremented. However, if the `count` variable is already zero, then it is not possible to decrement it further and the current task needs to wait. This is the point at which it is added to the list of waiting processes (`wait_list`) and the task state is set to UNINTERRUPTIBLE. Similar to the case of unlocking a spin lock, here also, if the `count` becomes non-zero from zero, we pick a process from the `wait_list` and set its state to RUNNING. Given that all of this is happening within the kernel, setting the task state is very easy. Needless to say, directly manipulating the state of a task is not possible to do at the user level. We need a system call for everything. However, in the kernel, we do not have such restrictions and thus these mechanisms are much faster.

5.3.4 The Lockdep Mechanism

We need a kernel lock validator that verifies whether there is a possibility of a deadlock or not. There is thus a need to have a deadlock avoidance mechanism that can be triggered just before acquiring a lock. In case, there is a potential of a deadlock, then the operation should either be stopped, delayed or allowed to proceed with a warning.

The way to orchestrate this is as follows. Whenever a lock is acquired, a call should be made to validate the lock acquire operation. There is a need to ensure that there is no possibility of deadlocks. This is precisely what the Linux kernel does. Refer to the function `lock_acquire` in kernel/locking/lockdep.c [Molnar, 2006]. Broadly speaking, the lockdep mechanism in the kernel is used to implement such a functionality.

It starts with doing a few trivial checks. First, it verifies that the lock depth is below a threshold. The lock depth is the number of locks that the current task has already acquired. There is a need to limit the number of locks that a thread is allowed to acquire at any point of time such that the overall complexity of the kernel and the lock validation system remains within bounds. Next, there is a need to validate the current set of lock acquisitions and check for the possibility of deadlocks. All kinds of lock acquisitions need to be validated: spinlocks, mutexes and reader-writer locks. The main aim is to *avoid* potential deadlock-causing situations.

Four kinds of states are defined: softirq − safe, softirq − unsafe, hardirq − safe and hardirq − unsafe. A softirq − safe state means that the lock was acquired while executing a *softirq*. At that point, softirqs were disabled. However, it is also possible to acquire a lock with the possibility of being preempted by a softirq. For example, this can happen if interrupts are enabled. In this case, the state of the lock acquisition will be softirq − unsafe. Hence, in the softirq − unsafe state, the thread can get preempted by a softirq handler.

In any unsafe state, it is possible that the thread can get preempted and an interrupt handler or softirq can run. This interrupt handler may try to acquire the same lock. If it has a higher priority, then this is clearly a deadlock-forming situation. This is because the high-priority handler will keep busy waiting. It will never be successful because the thread that has acquired the lock has a lower priority and will never get a chance to run. Such kind of a deadlock happens because a lock was acquired in the wrong context. It was acquired when interrupts or softirqs were enabled, which allowed higher priority entities to preempt the current thread. A deadlock can happen if they also request the same lock. This is a deadlock due to a *context inconsistency*.

Note that any softirq − unsafe state is hardirq − unsafe as well. This is because hard irq interrupt handlers have a higher priority as compared to softirq handlers. We define the states hardirq − safe and hardirq − unsafe analogously. These states will also be used to flag potential deadlock-causing situations.

We next *validate* the chain of lock acquire calls that have been made. We check for trivial bugs in the code that may lead to deadlocks. This is easy to detect. If there are two patterns of the form $A \rightarrow B$ and $B \rightarrow A$ in the chain of lock acquisitions, then it means that locks are not acquired in order. Whenever this happens, there is the possibility of a deadlock, and thus this pattern should be avoided. The lockdep mechanism can easily flag such risky patterns. This pattern is known as a *lock inversion*.

Next, let us look at problems created by context inconsistency. Consider all the lock acquisition operations for a given lock. Let us arrange them in a sequence. It should not contain a hardirq − unsafe lock acquisition and subsequently a hardirq − safe lock acquisition. An unsafe state allows an interrupt handler to execute. The interrupt handler can subsequently vie for the same lock. We don't know if this will be the case. However, we can try to be conservative and make a guess about whether there is a possibility of a deadlock. Our suspicion gets strong if the same lock is subsequently acquired in the hardirq − safe state. This means that it is meant to be used in an environment where interrupts are disabled. This means that most likely the lock cannot be safely used when interrupt handlers can preempt the thread that has just acquired the lock. Whenever a context inconsistency is detected, it means that a lock is most likely not being used consistently, and a deadlock could form.

Quick Detection of Cycles

Let us now look at the general case in which we have cyclic dependences. We need to create a global graph where each lock acquisition is a node, and if the process holding lock A waits to acquire lock B, then there is an arrow from A to B. If we have V nodes and E edges, then the time complexity of cycle detection is $O(V + E)$. This is quite slow. This needs to be done frequently – we need to check for cycles before acquiring every lock.

Lockdep uses a simple caching-based technique. Consider a chain of lock acquisitions, where the lock acquire calls can possibly be made by different threads. Given that the same kind of code sequences tend to repeat in the kernel code, we can cache a full sequence of lock acquisition calls. If the entire sequence is devoid of cycles, then we can deem the corresponding execution to be deadlock free. Hence, the brilliant idea here is as follows.

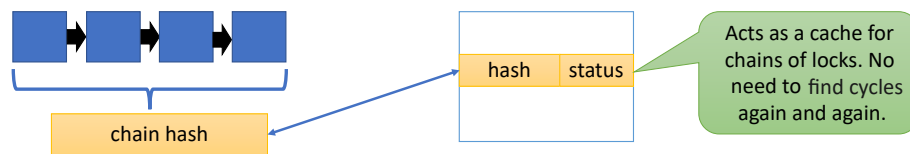Instead of checking for a deadlock on every lock acquire, we check for dead-

Figure 5.18: A hash table that stores an entry for every chain of lock acquisitions.

locks once in a while. We consider a long sequence (chain) of locks and compute a hash value for the entire chain. Subsequently, we create a graph out of the lock acquisitions and check for cycles. The hash table stores the "deadlock status" associated with such chains (see Figure 5.18). The key is the hash of the chain and the value is a Boolean variable. If there is a circular wait in the graph, the the value is 1, else it is 0.

This is a much faster mechanism for checking for deadlocks and the overheads are quite small. Note that if no entry is found in the hash table, then we either keep building the chain and try later, or we run a cycle detection algorithm immediately. The status of the chain of lock acquisitions is stored in the hash table after cycle detection.

5.3.5 The RCU (Read-Copy-Update) Mechanism

Managing concurrency in the kernel is a difficult problem. Let us now look at this problem from the angle of implementing ultra-efficient reader-writer locks and garbage collection [McKenney, 2003]. Let us introduce the RCU (read-copy-update) mechanism and draw a parallel with reader-writer locks, even though both are not strictly equivalent. Reader-writer locks can be thought of as the closest cousin of RCU in the world of concurrency.

> **Point 5.3.2**
>
> - RCU is typically used to access objects that are part of an "encapsulating data structure". A similar pattern was used to create generic tree and linked list nodes.
>
> - Instead of directly modifying a field in the object, we create a copy of the object in a private memory region, make all the updates, and then "publish the update". This is achieved by modifying pointers in the encapsulating data structure to point to the new copy of the object.
>
> - It is possible that there are live references to the previous version of the object, which are held by other threads. We need to wait for all the threads to finish their reads. This is known as the *grace period*. Reads are practically "zero overhead" operations. We do not need to acquire locks and there is no waiting.
>
> - Once the grace period has ended – all the readers are done reading – the object can be safely reclaimed and the space that was allocated

> to store it can be *freed*. RCU has a very fast mechanism for finding out when all the readers are done reading an object. In fact, this is the main challenge.

The process of allocating and freeing objects is the most interesting. Allocation is per se quite straightforward – we can use the regular *malloc* call. The object can then be used by multiple threads. However, freeing the object is relatively more difficult. This is because threads may have references to it. They may try to access the fields of the object after it has been freed. We thus need to free the allocated object only when no thread is holding a valid reference to it or is holding a reference but *promises* never to use it in the future. In C, it is always possible to arrive at the old address of an object using pointer arithmetic. However, let us not consider such tricky situations because RCU requires some degree of *disciplined programming*.

One may be tempted to use conventional reference counting, which is rather slow and complicated in a concurrent, multiprocessor setting. A thread needs to register itself with an object, and then it needs to deregister itself once it is done using it. Registration and deregistration increment and decrement the reference count, respectively. Any deallocation can happen only when the reference count reaches zero. This is a complicated mechanism. The RCU mechanism [McKenney, 2007] is comparatively far simpler. Simple implementations are fast implementations.

It needs to have the following features:

- There needs to be a way to deactivate pointers to a data structure such that they cannot be accessed subsequently.

- Without maintaining reference counts, it should be possible to figure out when an object (structure in C) can be safely freed.

- Once it is freed, its space can be reclaimed.

We shall primarily focus on the *freeing* part because it is the most difficult. Consider the example of a linked list (see Figure 5.19).
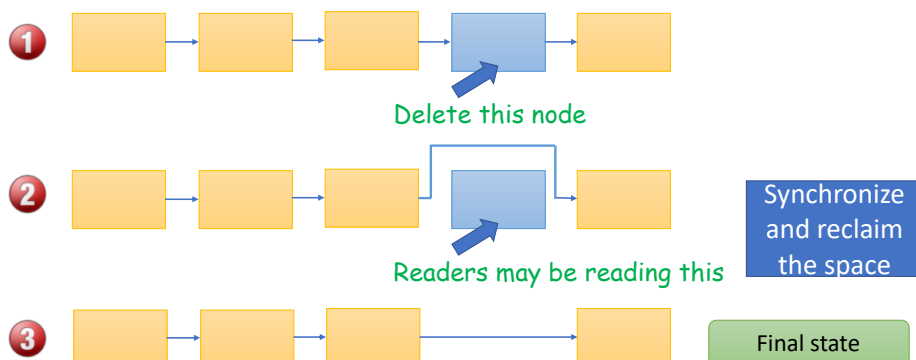
Figure 5.19: Deleting a linked list node (using RCU)

In this case, even though we delete a node from the linked list, other threads may still have references to it. The threads holding a reference to the node will not be aware that the node has been removed from the linked list. Hence, after deletion from the linked list we still cannot *free* the associated object.

Overview of the RCU Mechanism

The key idea in the RCU mechanism is to decouple the write(update), read and memory reclamation steps (include/linux/rcupdate.h). Let us quickly look at the basic RCU functions in Table 5.1. Their exact usage will be described later.

Call	Explanation	Action
`rcu_read_lock()`	Enter a read-side critical section	Disable preemption
`rcu_read_unlock()`	Exit a read-side critical section	Enable preemption
`synchronize_rcu()`	Marks the end of the reading phase	Wait until all readers finish
rcu_assign_pointer()	Assign a value to an RCU-protected pointer	Assignment + checks + memory barrier
rcu_dereference()	Convert an RCU-protected pointer to a regular kernel pointer	Read → memory barrier

Table 5.1: Key RCU functions (include/linux/rcupdate.h)

We define a few novel concepts. We define a read-side critical section that allows us to read an RCU-protected structure. However, unlike regular critical sections, here we just disable and enable preemption, respectively. The claim is that this is enough. It is also enough to find out when all the readers have stopped reading. We shall prove this subsequently. This is the crux of the RCU mechanism.

The main insight is that there is no need to maintain reference counts and wait till they reach zero. This is difficult to scale in a concurrent setting. Also waiting on a reference count requires busy waiting. We already know the problems in busy waiting such as cache line bouncing and doing useless work. This is precisely what we would like to avoid in the RCU mechanism. The `synchronize_rcu` call marks the end of the reading phase. It waits till all the readers have finished.

Writing is slightly different here – we create a copy of an object, modify it, and assign the new pointer to a field in the *encapsulating data structure*. Note that a pointer is referred to as *RCU protected*, when it can be assigned and dereferenced with special RCU-based checks (as we shall see later).

Using the RCU Read Lock

Listing 5.27: Example code that traverses a list within an RCU read context
source : include/linux/rcupdate.h

```
rcu_read_lock ();
list_for_each_entry_rcu(p, head, list) {
    t1 = p->a;
    t2 = p->b;
}
rcu_read_unlock ();
```

Listing 5.27 shows an example of traversing a linked list within an RCU context. We first acquire the RCU read lock using the `rcu_read_lock` call. This is actually quite simple – we just disable preemption by incrementing a per-CPU count stored in the `pcpu_hot` structure (refer to the discussion in Section 5.3.1). Then we iterate through the linked list and for each entry we transfer its contents to two temporary variables. Instead of this dummy code, we can have any other code snippet in its place. We are just showing a simple example here. Finally, we release the read lock using the `rcu_read_unlock` call. This does the reverse; it enables preemption by decrementing the same per-CPU count.

If this per-CPU count is 0, it means that preemption is disabled. Recall that if we disable preemption N times, then the count will become N. To re-enable preemption, we need to enable it N times. The key point here is that just enabling and disabling preemption is quite easy – all that we do is decrement or increment a count (resp.). There is no cache line bouncing and all the memory accesses are local to the CPU. This is very efficient in terms of performance.

Synchronizing the Readers

Listing 5.28: Replace an item in a list and then wait till all the readers finish

```
list_replace_rcu(&p->list, &q->list);
synchronize_rcu ();
kfree (p);
```

Listing 5.28 shows a piece of code that waits till all the readers complete. In this case, one of the threads calls the `list_replace_rcu` function that replaces an element in the list. It is possible that there are multiple readers who currently have a reference to the old element (`p->list`) and are currently reading it. We need to wait for all of them to finish the read operation. The only assumption that can be made here is that all of them are accessing the list in an RCU context – the code is wrapped between the RCU read lock and read unlock calls.

The function `synchronize_rcu` makes the thread wait for all the readers to complete. Once, all the readers have completed, we can be sure that the old pointer will not be read again. This is because the readers will check if the node pointed to by the pointer is still a part of the linked list or not. This is not enforced by RCU per se. Coders nevertheless have to observe such rules if they want to use RCU correctly.

After this we can free the pointer `p` using the `kfree` call.

Assigning an RCU-Protected Pointer

Listing 5.29: Assign an RCU-protected pointer
source : include/linux/rcupdate.h#L518

```
#define rcu_assign_pointer(p, v)
do {    \
    uintptr_t _r_a_p__v = (uintptr_t)(v);   \
    /* do some checking */  \
    if (__builtin_constant_p(v) && (_r_a_p__v) == (uintptr_t
        )NULL)   \
        WRITE_ONCE((p), (typeof(p))(_r_a_p__v));    \
    else   \
        smp_store_release(&p, RCU_INITIALIZER((typeof(p))
            _r_a_p__v));   \
} while (0)
```

Let us now look at the code for assigning a value to an RCU-protected pointer in Listing 5.29. The macro is written in the classical kernel style with a lot of underscores ☺. But it is not hard to understand. The idea is to basically perform the following operation `p = v`.

The first step is to change the type of the argument `v` to a pointer to an unsigned integer. The result is stored in the variable `_r_a_p__v`. If it is a NULL pointer or a built-in constant, then we can directly assign it to `p`. Consider the regular case now.

The kernel defines a separate memory area for RCU-protected variables. Whenever we assign a pointer to an RCU-protected pointer variable, we need to ensure that the pointer points to the RCU-specific memory region. The `RCU_INITIALIZER` macro performs this check and does necessary typecasting. The pointer in this case is `_r_a_p__v`.

Before assigning `_r_a_p__v` to `p`, we execute a memory barrier. This ensures that all the memory operations performed by the thread before it are fully visible to the rest of the threads. The assignment then happens at the very end. This is after the RCU check (see if the pointed object lies within the RCU region or not) and the memory barrier.

Listing 5.30: Implementation of the `list_replace_rcu` function
source : include/linux/rculist.h#L197

```
static inline void list_replace_rcu(struct list_head *old,
        struct list_head *new)
{
    new->next = old->next;
    new->prev = old->prev;

    /* new->prev->next = new */
    rcu_assign_pointer(list_next_rcu(new->prev), new);

    new->next->prev = new;
    old->prev = LIST_POISON2;
}
```

Let us now consider an example that uses the `rcu_assign_pointer` function in the context of the `list_replace_rcu` function (see Listing 5.30). In a doubly-linked list, we need to replace the `old` entry by `new`. We first start with setting

the **next** and **prev** pointers of **new** (make them the same as **old**). Note that at this point, the new node is not added to the list.

It is added when the **next** pointer of **new−>prev** is set to **new**. This is the key step that adds the **new** node to the list. We can think of this operation as the point of linearizability. This pointer assignment is done using an RCU function because we need to ensure proper memory ordering and visibility across cores.

Dereferencing a Pointer

Listing 5.31: Code to dereference an RCU pointer
source : include/linux/rcupdate.h#L459

```
#define __rcu_dereference_check(p, local, c, space) \
({ \
    typeof(*p) *local = (typeof(*p) *__force) READ_ONCE(p); \
    rcu_check_sparse(p, space); \
    ((typeof(*p) __force __kernel *)(local)); \
})
```

Let us now look at the code to deference a pointer to an RCU-protected variable (refer to Listing 5.31). The term "dereference" is actually a misnomer. The code converts an RCU-protected pointer to a regular kernel pointer. The reason is that the compiler and static code analysis tools incorporate a lot of checks to ensure that an RCU-protected pointer is always used in the right context. This means that it is used in functions that are meant to run exclusively in the RCU context. The advantage of this is that we can seamlessly track the readers and we will get to know exactly when there are no readers referencing an object. If we are not disciplined, then the RCU mechanism will not work. We cannot simply compute ***p**, where **p** is an RCU-protected pointer. This will make the current process a reader, even though it is not being tracked. Hence, the **rcu_dereference_check** function calls the function **__rcu_dereference_check**, which converts an RCU pointer to a regular kernel pointer. The regular kernel pointer can be used to access the object that it is pointing to in a conventional manner.

In Listing 5.31, we read the address stored in the pointer **p** and store it in a variable called **local**, which is a regular pointer. We next perform a memory check to ensure that **p** indeed points to a variable stored in the RCU region of the kernel's memory (**rcu_check_parse**). This means that **p** is an RCU-protected pointer. The last line returns the **local** pointer by forcefully converting it to a kernel pointer. All normal pointer-based operations can be performed on **local**. No compile-time errors will be thrown. Also all kernel code checking tools will allow such usage.

Note that the **READ_ONCE** macro includes a compile-time barrier. This means that the compiler cannot reorder the read operation. However, this does not require a hardware memory barrier (fence). The idea here is to rely on some of the memory orderings guaranteed by the underlying x86 hardware. This is a standard pattern in the kernel code. Whenever there is no requirement for adding a hardware-level fence operation, it is not added. x86 is not a very weak memory model that allows all kinds of reorderings. It primarily reorders writes and subsequent reads to different addresses. The rest of the orderings are

preserved. Sometimes this property can be used to eliminate the requirement for fence operations. The caveat here is that the compiler should not do any reordering, which in this case is being explicitly prevented.

Listing 5.32: Example that uses the RCU dereference operation
source : include/linux/rculist.h#L688

```
#define __hlist_for_each_rcu(pos, head)           \
    for (pos = rcu_dereference(hlist_first_rcu(head)); \
        pos;                                       \
        pos = rcu_dereference(hlist_next_rcu(pos)))
```

Listing 5.32 shows an example of using `rcu_dereference`. It shows a *for* loop. The iteration starts at the `head` node that is dereferenced in an RCU context. In every iteration, it proceeds to the next node on the list and finally the iteration stops when the current node becomes NULL.

Correctness of the RCU-based Mechanism

Let us now prove the correctness of the RCU mechanism. Consider the classical list replace algorithm, where we replace an element with another element. In this case, Thread j (running on Core j) that removed a list element needs to also *free* it. Before freeing it, Thread j needs to wait till all the readers that are currently reading it complete their reads.

Given that we are not using atomic variables to maintain reference counts, we need to prove how our simple preemption-based mechanisms can guarantee the same. If there is a concurrent read, then it must have acquired the read lock. Assume that this read operation is executing on Core i. On that core, preemption is disabled because the read lock on Core i is currently acquired by the reading thread.

All that Thread j needs to do is send an inter-processor interrupt to each core and run a task on it. When preemption is disabled (within the RCU read-side critical section), no task can run on the core. We need to wait for preemption to be enabled again. Preemption will be re-enabled when Thread i leaves the read-side critical section (calls the read unlock call). Subsequently, the small task can run on Core i. The fact that this task is running means that the reader has finished. Note that the assumption is that the next read operation on Core i will not be able to see the element that was replaced because it is not there in the list anymore.

Now, if we can run such tasks on every core (CPU in Linux), then it means that all the *readers have left the RCU read-side critical section*. The code is shown in Listing 5.33.

Listing 5.33: Code to run tasks on each CPU

```
foreach_cpu (cpu)
    run_curr_task_on_cpu (cpu);
```

At this point the object can be freed, and its space can be reclaimed. This method is *simple* and *slow*.

RCU in all its Glory

The basic scheme of running a task on each CPU is a slow and simple scheme, which is quite inefficient in terms of performance. Let us consider more realistic implementations. There are two implementations: tiny RCU and tree RCU [Community,]. The former is for non-preemptible uniprocessors, and thus is not very relevant as of today. The latter is the default implementation on multicore processors. Let us look at its design.

Let us start out by defining the *grace period*, which is the time between updating an object and reclaiming its space. The grace period should be large enough for all the readers to finish their reads (refer to Figure 5.20).
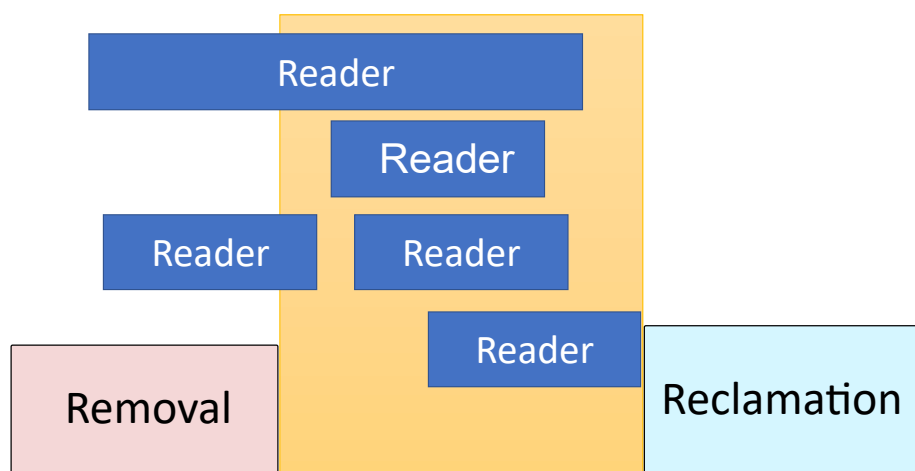
Figure 5.20: Removal and reclamation of an object (within the RCU context)

This figure shows a thread that first initiates the "removal" phase. This is when the object is removed from its containing data structure. There could be live references to it that are held by other threads, which are concurrently reading it. We wait for all the readers to complete (grace period to end). After releasing the RCU read lock, threads enter the *quiescent state* [McKenney, 2008]. They don't read the object anymore. At the end of the grace period, all the threads are in the quiescent state. Then we can free the object (reclamation).

Let us now understand when the grace period (from the point of view of a thread) ends and the period of quiescence starts. One of the following conditions needs to be satisfied.

1. When a thread blocks: If there is a restriction that blocking calls are not allowed in an RCU read block, then if the thread blocks we can be sure that it is in the quiescent state.

2. If there is a switch to user-mode execution, then we can be sure that the kernel has finished executing its RCU read block.

3. If the kernel enters an idle loop, then also we can be sure that the read block is over.

4. Finally, if we switch the context to another kernel thread, then also we can be sure that the quiescent state has been reached.

Whenever any of these conditions is true, we set a bit that indicates that the CPU is out of the RCU read block – it is in the quiescent state. The reason that this enables better performance is as follows. There is no need to send costly inter-processor interrupts to each CPU and wait for a task to execute. Instead, we infer quiescence based on the state of the execution of the thread. The moment a thread leaves the read block, the CPU enters the quiescent state and this fact is immediately recorded by setting a corresponding per-CPU bit. Note the following: this action is off the critical path and there is no shared counter.

Once all the CPUs enter a quiescent state, the grace period ends and the object can be reclaimed. Hence, it is important to answer only one question when a given CPU enters the quiescent state: Is this the last CPU to have entered the quiescent state? If, the answer is "Yes", then we can go forward and declare that the grace period has ended. The object can then be reclaimed. This is because we can be sure that no thread holds a valid reference to the object (see the answer to Question 5.3.5).

Question 5.3.1

What if there are more threads than CPUs? It is possible that all of them hold references to an object. Why are we maintaining RCU state at the CPU level?

Answer: We assume that whenever a thread accesses an object that is RCU-protected, it is accessed only within an RCU context (within a read block). Furthermore, a check is also made within the read block to ensure that it is a part of the encapsulating data structure. It cannot access the object outside the RCU context. Now, once a thread enters an RCU read block, it cannot be preempted until it has finished executing the read block.

It is not possible for the thread to continue to hold a reference and use it. This is because it can be used once again only within the RCU context, and there it will be checked if the object is a part of its containing data structure. If it has been removed, then the object's reference has no value.

For a similar reason, no other thread running on the CPU can access the object once the object has been removed and the quiescent state has been reached on the CPU. Even if another thread runs on the CPU, it will not be able to access the same object because it will not find it in the encapsulating data structure.

Because we do not allow multiple threads to preempt each other in an RCU read block, maintaining state at the CPU level is sufficient. It just ensures that the currently running thread has entered a period of quiescence. Subsequent threads running on the CPU will not be able to access the object that was removed anyway.

Tree RCU

Let us now suggest an efficient method of managing the quiescent state of all CPUs. The best way to do so is to maintain a tree. Trees have natural parallelism; they avoid centralized state.

`struct rcu_state` is used to maintain quiescence information across the cores. Whenever the grace period ends (all the CPUs are quiescent at least once), a callback function may be called. This will let the writer know that the object can be safely reclaimed.
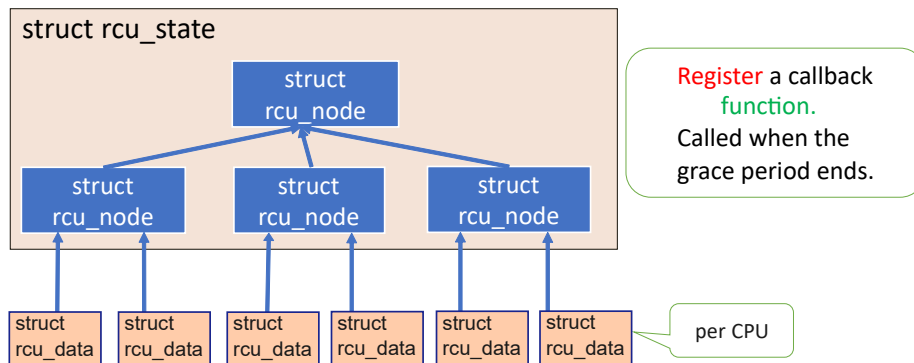

Figure 5.21: Tree RCU

RCU basically needs to set a bit in a per-CPU data structure (`struct rcu_data`) to indicate the beginning of a quiescent period. This is fine for a simplistic implementation. However, in practice there could be nested RCU calls. This means that within an RCU read block, the kernel may try to enter another RCU read block. In this case, the CPU enters the quiescent state when the execution has moved past all the RCU read blocks and is currently not executing any code that is a part of an RCU read-side critical section. The CPU could have been executing in an idle state or there could have been a context switch. It is often necessary to distinguish between the reasons for entering a period of quiescence. Hence, we need a more complex multibit mechanism that indicates that the CPU is "quiescent" and the reason for entering the quiescent state. Hence, in modern kernels a slightly more complex mechanism is used. It is known as the *dynticks* mechanism that incorporates a bunch of counters. One of those counters is checked to see if it is even or odd. If it is even, then it means that the CPU is *quiescent* otherwise it means that it is not. However, there are other pieces of information as well that we need to maintain such as the number of grace periods that have elapsed, the nature of the quiescent state, scheduler activations, interrupt processing and so on. For the sake of the discussion in this section, let us assume that each `rcu_data` structure (corresponding to a CPU) stores just one bit – quiescent or not.

We can then organize this information in an augmented tree, and create a tree of nodes. The internal nodes are of type `struct rcu_node` as shown in Figure 5.21. To find out if the entire system is quiescent or not, we just need to check the value of the root of the tree. This makes this process very fast and efficient.

Preemptible RCU

Sadly, RCU stops preemption and migration when the control is in an RCU block (read-side critical section). This can be detrimental to real-time programs as they come with strict timing requirements and deadlines. In real-time versions of Linux, there is a need to have a preemptible version of RCU where preemption is allowed within an RCU read-side block. Even though doing this is a good idea for real-time systems, it can lead to many complications.

In classical RCU, read-side critical sections had almost zero overhead. Even on the write-side all that we had to do is read the current data structure, make a copy, make the changes (update) and add it to the encapsulating data structure (such as a linked list or a tree). The only challenge was to wait for all the outstanding readers to complete, which has been solved very effectively.

Here, there are many new complications if we make critical sections preemptible. If there is a context switch in the middle of a read block, then the read-side critical section gets "artificially lengthened". We can no more use the earlier mechanisms for detecting quiescence. In this case, whenever a process enters a read block, it needs to register itself, and then it needs to deregister itself when it exits the read block. Registration and deregistration can be implemented using counter increments and decrements, respectively. The `rcu_read_lock` function needs to increment a counter and the `rcu_read_unlock` function needs to decrement a counter. These counters are now a part of a process's context, not the CPU's context (unlike classical RCU). This is because we may have preemption and subsequent migration. It is also possible for two concurrent threads to run on a CPU that access RCU-protected data structures. Note that this was prohibited earlier. We waited for a read block to completely finish before running any other thread. In this case, two read blocks can run concurrently (owing to preemption). Once preempted, threads can also migrate to other CPUs. Hence, counters can no more be per-CPU counters. State management thus becomes more complex. Summary: This mechanism enables real-time execution and preemption at the cost of making RCU slower and more complex.

5.4 Scheduling

Scheduling is one of the most important activities performed by the kernel. It is a major determinant of the overall system's responsiveness and performance.

5.4.1 Space of Scheduling Problems

Scheduling per se is an age-old problem. There are a lot of variants of the basic scheduling problem. Almost all of them have the same basic structure, which is as follows. We have a bunch of jobs that needs to be scheduled on a set of cores. Each job has a start time (or arrival time) and a duration (time it takes to execute). The task is to schedule the set of jobs on all the cores while ensuring some degree of optimality at the same time. The definition of an *optimal schedule* here is not very obvious because many criteria for optimality exist. Also, for a single core we may use one criteria and a different one for a multicore processor.

Note that it is important to differentiate between a *task* and a *job* here. A *task* is a process that owns resources, can get swapped out and can be migrated across cores. It can spawn many jobs, where each job is a quantum of CPU activity. For example, if there is a periodic task, it spawns a new job once every period. Each such job is an independently schedulable entity. Hence, we shall use the term "jobs" in this section. A task is a bigger entity that corresponds to a process, which might create many jobs in its lifetime.

Figure 5.22: Example of a set of jobs that are waiting to be scheduled

Figure 5.22 shows an example where we have a bunch of jobs that need to be scheduled. In this case, we assume that the time that a job needs to execute (processing time) is known a priori (shown in the figure).

Objective Functions

Mean Completion Time

In single-core processors, we typically use the mean completion time as the objective function that needs to be minimized (refer to Figure 5.23). The completion time of a job is the time it takes to complete. We can compute it by subtracting its arrival time from its completion time. In this interval, it may get preempted several times. Ultimately, when the job completes, the completion time is recorded. The objective function here is to minimize the mean completion time across all the jobs. If the number of jobs is known (which it often is), then this objective is the same as the total completion time.

The mean completion time determines the *responsiveness* of the system. If a scheduling system delays a lot of jobs, it will have an adverse mean completion time.
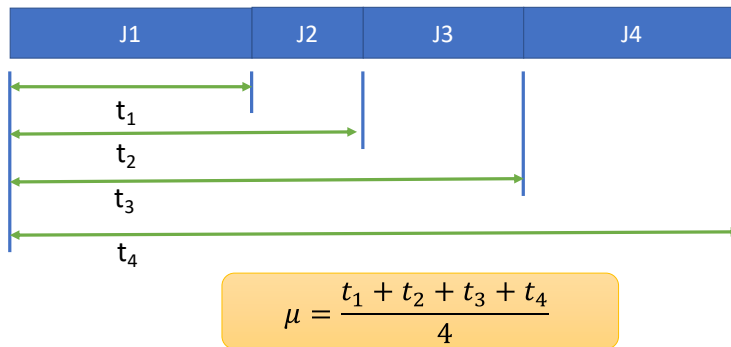
Figure 5.23: Mean completion time $\mu = \sum_i t_i/n$

Makespan

Next, let us look at optimality criteria in the space of multicore scheduling. The aim is to minimize the *makespan*. The makespan is the time duration between the time at which scheduling starts and the time at which the last job is completed. This is basically the time that is required to finish the entire set of jobs on a parallel machine.

Issues with the Objective Functions

Let us now try to punch a set of holes into the definitions of the mean completion time and makespan that we just provided. We are assuming that the time that a job takes to execute is known. This is seldom the case in a practical real-life application, unless it is a very controlled system like an embedded real-time system.

Nevertheless, for theoretical and mathematical purposes, such assumptions are made very frequently. In many cases, it is indeed possible to accurately estimate the duration of the execution using prior history. Hence, this assumption may not be that impractical all the time.

Preemption and Arrival Times

The next point to consider is whether the jobs are preemptible or not. If they are, then the problem actually becomes quite simple most of the time. Whereas, if they aren't, then the problem often becomes far more complex. Many variants of the scheduling problem with this restriction are NP-complete. The reasons are obvious. For preemptible jobs, we can arbitrarily split them and execute the remaining portion either later or on another core. This allows for much more flexibility in job scheduling.

Along with preemptibility, we also need to look at the issue of arrival times. Some simple models of scheduling assume that the arrival time is the same for all the jobs. This means that all the jobs arrive at the same time, which we can assume to be $t = 0$. In another model, we assume that the arrival times are not the same for all the jobs. Here again, there are two types of problems. In one case, the jobs that will arrive in the future are known. In the other case, we have no idea – jobs may arrive unannounced at any point of time.

We can thus observe that the problem of scheduling is a very fertile ground for proposing and solving optimization problems. We can have a lot of constraints, settings and objective functions.

To summarize, we have said that in any scheduling problem, we have a list of jobs. Each job has an arrival time, which may either be equal to 0 or some other time instant. Next, we typically assume that we know how long a job shall take to execute. Then in terms of constraints, we can either have preemptible jobs or we can have non-preemptible jobs. The latter means that the entire job needs to execute in one go without any other intervening jobs. Given these constraints, there are a couple of objective functions that we can minimize. One would be to minimize the makespan, which is basically the time from the start of scheduling till the time it takes for the last job to finish execution. Another objective function is the average completion time, where the completion time is again defined as the time at which a job completes minus the time at which it arrived (measure of responsiveness).

For scheduling such a set of jobs, we have a lot of choices. We can use many simple algorithms, which in some cases can also be proven to be optimal. Let

us start with the *random* algorithm. It randomly picks a job and schedules
it on a free core. There is a lot of work that analyzes the performance of
such algorithms and many times such random choice-based algorithms in fact
perform reasonably well. However, in modern systems, it is advisable to take a
more serious approach. It is best to use algorithms that are provably optimal
or have been proven to be near-optimal either theoretically or empirically.

KSW Model

Let us now introduce a more formal way of thinking and introduce the Karger-
Stein-Wein (KSW) model [Karger et al., 1999]. It provides an abstract or generic
framework for all scheduling problems. It essentially divides the space of prob-
lems into large classes and finds commonalities between problems that belong
to the same class. Specifically, it requires three parameters: α, β and γ.

The first parameter α determines the machine environment. It specifies the
number of jobs and the processing time of each job. The second parameter
β specifies the constraints. For example, it specifies whether preemption is
allowed or not, whether the arrival times are the same or are different, whether
the jobs have dependencies between them or whether there are job deadlines.
A dependence between a pair of jobs can exist in the sense that we can specify
that job J_1 needs to complete before J_2. Note that in real-time systems, jobs
come with deadlines, which basically means that jobs have to finish before a
certain time. *Deadlines* are thus another type of constraint.

Finally, the last parameter γ is the optimality criterion. We have already
discussed the average mean completion time and makespan criteria. We can
also define a *weighted* completion time – a weighted mean of completion times.
Here a weight corresponds to a job's priority. It is easy to observe that the
mean completion time metric is a special case of the weighted completion time
metric – all the weights are equal to 1. Now, let the completion time of job i be
C_i. The cumulative completion time can be equivalent to the mean completion
time if the number of jobs is a constant. We can represent this criterion as ΣC_i.
The makespan is C_{max} (maximum completion time of all jobs).

We can consequently have a lot of scheduling algorithms for every scheduling
problem, which can be represented using the 3-tuple $\boxed{\alpha \mid \beta \mid \gamma}$ as per the KSW
formulation.

The most popular scheduling algorithms are quite simple, and are also prov-
ably optimal in some scenarios. We shall also introduce a bunch of settings
where finding the optimal schedule is an NP-complete problem [Cormen et al.,
2009]. There are good approximation algorithms for solving such problems.

5.4.2 Single Core Scheduling

The Shortest Job First Algorithm

Let us define the problem $1 \mid\mid \Sigma C_j$ in the KSW model. We are assuming
that there is a single core. The objective function is to minimize the sum of
completion times (C_j). Note that minimizing the sum of completion times is
equivalent to minimizing the mean completion time because our assumption is
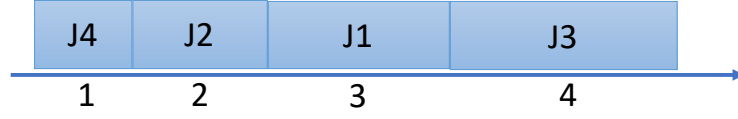that the number of tasks is known a priori and is a constant.

Figure 5.24: Shortest job first scheduling

The claim is that the SJF (shortest job first) algorithm is optimal in this case (example shown in Figure 5.24). Let us outline a standard approach for proving that a scheduling algorithm is optimal with respect to the criterion that is defined in the KSW formulation. Here we are minimizing the mean completion time.

Let the SJF algorithm be algorithm A. Assume that another algorithm A' is optimal. There must be a pair of jobs j and k such that j immediately precedes k and the processing time (execution time) of $j > k$. This means $p_j > p_k$. Note that such a pair of jobs will not exist in the schedule produced by algorithm A. Assume p_j started at time t. Let us exchange jobs j and k with the rest of the schedule generated by A' remaining the same. Let us assume that this new schedule is produced by another algorithm A''.

Next, let us evaluate the contribution to the cumulative completion time by jobs j and k in the schedule generated by algorithm A'. It is $(t+p_j)+(t+p_j+p_k) = 2t+2p_j+p_k$. Let us evaluate the contribution of these two jobs in the schedule produced by A''. It is $(t+p_k)+(t+p_j+p_k) = 2t+2p_k+p_j$. Given that $p_j > p_k$, we can conclude that the schedule produced by algorithm A' is longer (larger cumulative completion time). This can never be the case because we have assumed A' to be optimal. We thus have a contradiction here because A'' appears to be better than A', which violates our assumption. This leads to a contradiction.

Hence, A' or any algorithm that violates the SJF order cannot be optimal. Thus, algorithm A (SJF) is optimal.

Weighted Jobs

Let us now define the problem where weights are associated with jobs. It will be $1 \parallel w_j C_j$ in the KSW formulation. If $\forall j, w_j = 1$, we have the classical unweighted formulation for which SJF is optimal.

For the weighted version, let us schedule jobs in descending order of (w_j/p_j). Clearly, if all $w_j = 1$, this algorithm is the same as SJF. We can use the same exchange-based argument to prove that using (w_j/p_j) as the job priority yields an optimal schedule.

EDF Algorithm

Let us next look at the EDF (Earliest Deadline First) algorithm. It is one of the most popular algorithms in real-time systems. There are different flavors of the EDF algorithm. Each one has a different problem formulation. In real-time systems, typically versions of EDF with periodic tasks are considered. In this section, let us consider a slightly different formulation where tasks are not periodic. However, we assume that we can preempt tasks with zero overhead.

Here, each job is associated with a distinct non-zero arrival time and deadline. Let us define the *lateness* as $\langle completion\_time \rangle$ - $\langle deadline \rangle$. The formulation is as follows:

$$1 \mid r_i, dl_i, pmtn \mid L_{max}$$

We are still considering a single core machine. The constraints are on the arrival time and deadline. r_i represents the fact that job i is associated with arrival time r_i – it can start only after it has arrived (r_i). Furthermore, it does not matter if the information about a job's arrival is known a priori or not. In other words, jobs can thus arrive at any point of time (dynamically). The dl_i constraint indicates that job i has deadline dl_i associated with it – it needs to complete before it. Preemption is allowed (*pmtn*). We wish to minimize *maximum lateness* (L_{max}). This means that we would like to ensure that jobs complete as soon as possible, with respect to their deadline. Note that in this case, we care about the maximum value of the lateness, not the mean value. This means that we don't want any single job to be delayed significantly.

The algorithm schedules the job whose deadline is the earliest. Assume that a job is executing, and a new job arrives that has an earlier deadline. Then the currently running job is swapped out, and the new job that now has the earliest deadline executes.

The proof is on similar lines and uses exchange-based arguments (refer to [Mall, 2009]).

We shall revisit EDF in the context of real-time scheduling in Section 5.5.2. There we will consider periodic tasks where the deadline is the same as the period. We will be able to make stronger guarantees with respect to schedulability.

SRTF Algorithm

Let us continue our journey and consider another problem of a similar variety: $1 \mid r_i, pmtn \mid \Sigma C_i$. Preemption is allowed and jobs can arrive at any point of time. We aim to minimize the mean/cumulative completion time.

In this case, the most optimal algorithm is SRTF (*shortest remaining time first*). For each job, we keep a record of the time that is left for it to finish its execution. We sort this list in ascending order and choose the job that has the shortest amount of time left. If a new job arrives, we compute its remaining time, and if that number happens to be the lowest, then we preempt the currently running job and execute the newly arrived job.

We can prove that this algorithm minimizes the mean (cumulative) completion time using a similar exchange-based argument.

Some NP-Completeness and Impossibility Results

There is a deep connection between preemption, arrival times and finding optimal schedules. In general, it is more difficult to design algorithms that do not have preemptible tasks. If jobs are preemptible, things are typically easier. For those who are theoretically minded, they will quickly appreciate why this is the case. It is possible to solve linear programming that is continuous domain optimization in polynomial time. However, solving integer-linear programming with integer constraints is NP-complete. This is because we have a

combinatorial explosion. Something similar happens in the case of scheduling problems.

However, preemption does not have an effect when all the jobs arrive at $t = 0$ for some common problems. Specifically, the quality of the schedule is not affected by whether preemption is allowed or not [Karger et al., 1999]. In does not make any difference insofar as the following optimality criteria are considered: ΣC_i or L_{max}. We have seen some of these problems. Consider $1 \parallel \sigma C_j$. In this case, SJF and SRTF produce the same schedule. Hence, preemption does not matter. Next, consider $1 \mid pmtn, dl_i \mid L_{max}$. Here, EDF will always produce the same schedule regardless of preemption. Hence, if all the jobs arrive at $t = 0$, and we consequently have full information about all the tasks, for such formulations preemption is not a virtue.

Next, let us discuss a few NP-complete problems in this space. Here, we assume that jobs have distinct non-zero arrival times.

- $1 \mid r_i \mid \Sigma C_i$: In this case, preemption is not allowed and jobs can arrive at any point of time. There is much less flexibility in this problem setting. This problem is provably NP-complete.

- $1 \mid r_i \mid L_{max}$: This problem is similar to the former. Instead of the average (cumulative) completion time, we have *lateness* as the objective function.

- $1 \mid r_i, pmtn \mid \Sigma w_i C_i$: This is a preemptible problem that is a variant of $1 \mid r_i, pmtn \mid \Sigma C_i$, which has an optimal solution – SRTF. The only addition is the notion of the weighted completion time. It turns out that for generic weights, this problem becomes NP-complete.

We thus observe that making a small change to the problem renders it NP-complete. This is how sensitive these scheduling problems are.

Practical Considerations

All the scheduling problems that we have seen assume that the job execution (processing) time is known. This may be the case in really well-characterized and constrained environments. However, in most practical settings, the job duration is not known.

$$t_n = \alpha t_{n-1} + \beta t_{n-2} + \gamma t_{n-3}$$
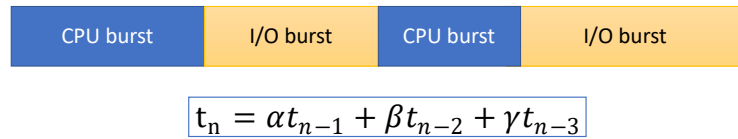
Figure 5.25: CPU and I/O bursts

Figure 5.25 shows a typical scenario. Any task typically cycles between two bursts of activity: a CPU-bound burst and an I/O burst. The task typically does a fair amount of CPU-based computation, and then makes a system call. This initiates a burst where the task waits for some I/O operation to complete. We enter an I/O bound phase in which the task typically does not actively execute. We can, in principle, treat each CPU-bound burst as a separate job.

Each task thus yields a sequence of jobs that have their distinct arrival times. The problem reduces to predicting the length of the next CPU burst.

We can use classical time-series methods to predict the length of the CPU burst. We predict the length of the nth burst t_n as a function of $t_{n-1}, t_{n-2} \ldots t_{n-k}$. For example, t_n could be described by the following equation:

$$t_n = \alpha t_{n-1} + \beta t_{n-2} + \gamma t_{n-3} \tag{5.1}$$

Such approaches that are rooted in time series analysis often tend to work and yield good results because the length of the CPU bursts have a degree of temporal correlation. The recent past is a good predictor of the immediate future. Using these predictions, the algorithms listed in the previous sections such as EDF, SJF and SRTF can be used. At least some good solutions can be realized.

Let us consider the case when we have a poor prediction accuracy. We can rely on simple, classical and intuitive methods as we shall describe next.

Conventional Algorithms

We can always make a random choice, however, that is definitely not desirable here. Something that is much more fair is a simple FIFO (first-in-first-out) algorithm. To implement it, we just need a queue of jobs. It guarantees the highest priority to the job that arrived the earliest. A problem with this approach is the "convoy effect". A long-running job can delay a lot of smaller jobs. They will get unnecessarily delayed. If we had scheduled them first, the average completion time would have been much lower.

We can alternatively opt for round-robin scheduling. We schedule a job for one time quantum. After that we preempt the job and run another job for one time quantum, so on and so forth. This is at least fairer to the smaller jobs – they complete sooner.

There is thus clearly a trade-off between the priority of a task and system-level fairness. If we boost the priority of a task, it may be unfair to other tasks (refer to Figure 5.26).

Figure 5.26: Fairness vs priority

We have discussed the notion of priorities in Linux (in Section 3.1.6). If we have a high-priority task running, we penalize other low-priority tasks. A need

for fairness thus arises. Many other operating systems such as Windows use other types of fairness metrics. For example, Windows boosts the priority of foreground processes by $3\times$. This means that if we start interacting with an application, its priority gets boosted.

Queue-based Scheduling
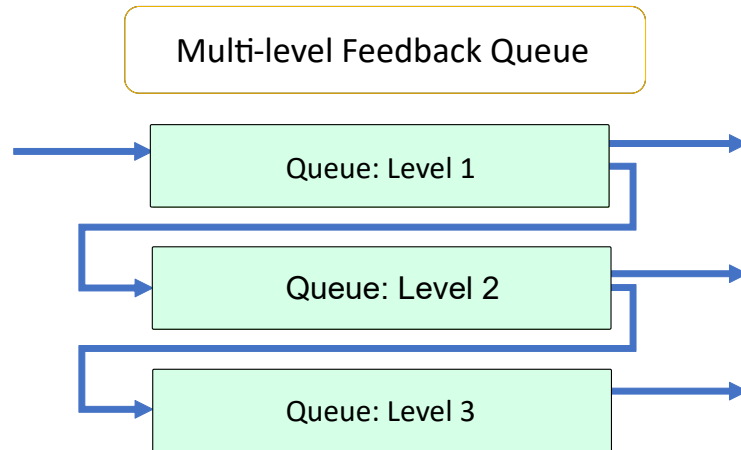
Figure 5.27: Multi-level feedback queue

A standard method of scheduling tasks that have different priorities is to use a multilevel feedback queue as shown in Figure 5.27. Different queues in this composite queue are associated with different priorities. We start with the highest-priority queue and start scheduling tasks using any of the algorithms that we have studied. If empty cores are still left, then we move down the priority order of queues: schedule tasks from the second-highest priority queue, third-highest priority queue and so on. Again note that we can use a different scheduling algorithm for each queue. They are independent in that sense.

Depending upon the nature of the task and for how long it has been waiting, tasks can migrate between queues. To provide fairness, tasks in low-priority queues can be moved to high-priority queues. If a background task suddenly comes into the foreground and becomes interactive, its priority needs to be boosted, and the task needs to be moved to a higher priority queue. On the other hand, if a task stays in the high-priority queues for a long time, we can demote it to ensure fairness. Such movements ensure both high performance and fairness.

5.4.3 Multicore Scheduling

Let us now come to the issue of multicore scheduling. The big picture is shown in Figure 5.28. We have a global queue of tasks that typically contains newly created tasks or tasks that need to be migrated. A dispatcher module sends the tasks to different per-CPU task queues. Theoretically, it is possible to have different scheduling algorithms for different CPUs. However, this is not a common pattern.
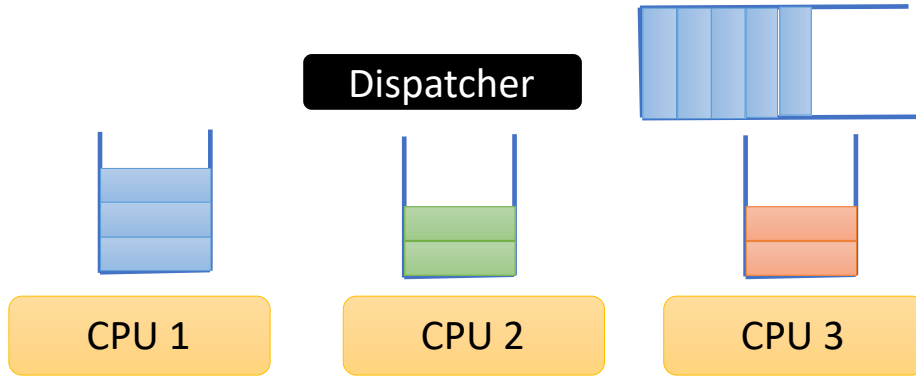
Figure 5.28: Multicore scheduling

Theoretical Results in Multicore Scheduling

The key objective function here is to minimize the makespan C_{max}. This is the maximum completion time across all cores – the time at which an ensemble of jobs finishes. Preemptible variants are in general easier to schedule. This is because such a problem is a continuous version of the scheduling problem and is similar in principle to linear programming that has simple polynomial time solutions. On the other hand, non-preemptive versions are far harder to schedule and are often NP-complete. These problems are similar to the knapsack, partition or bin packing problems (see [Cormen et al., 2009]), which are quintessential NP-complete problems. A problem in NP (nondeterministic polynomial time) can be *verified* in polynomial time if a solution is presented. These are in general *decision problems* that have yes/no answers. Note that the set of NP-complete problems are the hardest problems in NP. This means that if we can solve them in polynomial time, then we have polynomial time solutions for all the problems in NP.

Let us consider a simple version of the multicore scheduling problem: $P \mid pmtn \mid C_{max}$. Here, we have P processors and preemption is enabled. The solution is to simply and evenly divide the work between the cores and schedule the jobs. Given that every job can be split arbitrarily, scheduling in this manner becomes quite simple. In this case, $C_{max} = \frac{\sum C_i}{n}$.

However, the problem $P \mid\mid C_{max}$ where preemption is not enabled is NP-complete. Jobs cannot be split, and this is the source of all our difficulties.

To prove the NP-completeness of such problems, there is a standard method. We consider a problem that is known to be NP-complete. Next, we map every instance of this known NP-complete problem to a corresponding instance of a scheduling problem. The NP-complete problems that are typically chosen are the bin packing and partition problems. Once we have mapped bin packing to a scheduling problem, it is clear that if we can solve the scheduling problem in polynomial time, then we can solve all instances of the bin packing problem in polynomial time. Hence, the scheduling problem is harder than the bin packing problem. If the bin packing problem is NP complete, then so is the scheduling problem.

Point 5.4.1

Bin Packing Problem: We have a finite number of bins, where each bin has a fixed capacity S. There are n items. The size of the ith item is s_i. We need to pack the items in bins without exceeding any bin's capacity. The objective is to minimize the number of bins and find a mapping between items to bins such that we use the least number of bins.

Point 5.4.2

Set Partition Problem: Consider a set $\mathcal{S}$ of numbers. Find a subset whose sum is equal to a given value T.

Both these classical NP-complete problems – bin packing and set partition – assume that an item or set element cannot be split. Scheduling without preemption has a similar character.

List Scheduling

Let us consider one of the most popular non-preemptive scheduling algorithms in this space known as *list scheduling*. We maintain a list of ready jobs. They are sorted in descending order according to some priority scheme. When a CPU becomes free, it fetches the highest priority job from the list. In case, it is not possible to execute it, then the CPU walks down the list and finds another job to execute. The only condition here is that we cannot return without a job if the list is non-empty. This means that there is no deliberate idling. If a CPU wants to run something and the queue of jobs is non-empty, then the CPU cannot remain idle. Moreover, all the machines are considered to be identical in terms of computational power.

Let us take a deeper look at the different kinds of priorities that we can use. We can order the jobs in descending order of arrival time or job processing time. We can also consider dependencies between jobs. In this case, it is important to find the longest path in the graph (jobs are nodes and dependency relationships are edges). The *longest path* is known as the critical path. The critical path often determines the overall makespan of the schedule assuming we have adequate compute resources. This is why in almost all scheduling problems, a lot of emphasis is placed on the critical path. We always prefer scheduling jobs on the critical path as opposed to jobs off the critical path. We can also consider attributes associated with nodes in this graph. For example, we can set the priority to be the out-degree (number of outgoing edges). If a job as a high out-degree, then it means that a lot of other jobs are dependent on it. Hence, if this job is scheduled, many other jobs will get benefited – they will have one less dependency.

It is possible to prove that list scheduling is near-optimal using theoretical arguments [Graham, 1969]. Consider the problem $P \parallel C_{max}$. Let the makespan (C_{max}) produced by an optimal scheduling algorithm OPT have a length C^*. Let us compute the ratio of the makespan produced by list scheduling and C^*. Our claim is that regardless of the priority that is used, we are never worse off by a factor of 2. This is assuming that there is no deliberate idling.

> **Theorem 5.4.1** Makespan
>
> Regardless of the priority scheme, $C_{max}/C^* \leq 2 - \frac{1}{m}$. C_{max} is the length of the makespan of the schedule produced by list scheduling. There are m CPUs.

Proof: Let there be n jobs and m CPUs. Let the execution times of the jobs be $p_1 \ldots p_n$, and let job k (execution time p_k) complete the last. Assume it started at time t. Then $C_{max} = t + p_k$.

Given that there is no idleness in list scheduling, we can conclude that till t all the CPUs were 100% busy. This means that if we add all the work done by all the CPUs till point t, it will be mt. This comprises the execution times of a subset of jobs that does not include job k (one that completes the last). We thus arrive at the following inequality.

$$\sum_{i \neq k} p_i \geq mt$$

$$\Rightarrow \sum_i p_i - p_k \geq mt$$

$$\Rightarrow t \leq \frac{1}{m} \sum_i p_i - \frac{p_k}{m} \tag{5.2}$$

$$\Rightarrow t + p_k = C_{max} \leq \frac{\sum_i p_i}{m} - \frac{p_k}{m} + p_k$$

$$\Rightarrow C_{max} \leq \frac{\sum_i p_i}{m} + p_k \left(1 - \frac{1}{m}\right)$$

Now, $C^* \geq p_k$ and $C^* \geq mean(p_i)$. These follow from the fact that jobs cannot be split across CPUs (no preemption) and we wait for all the jobs to complete. We thus have,

$$C_{max} \leq \frac{\sum_i p_i}{m} + p_k \left(1 - \frac{1}{m}\right)$$

$$\leq C^* + C^* \left(1 - \frac{1}{m}\right) \tag{5.3}$$

$$\Rightarrow \frac{C_{max}}{C^*} \leq 2 - \frac{1}{m}$$

∎

Equation 5.3 shows that in list scheduling, a bound of $2 - 1/m$ is always guaranteed with respect to the optimal makespan. Note that this value is independent of the number of jobs and the way in which we assign priorities as long as we avoid deliberate CPU idling. Graham's initial papers in this area (see [Graham, 1969]) started a flood of research work in this area. People started looking at all kinds of combinations of constraints, priorities and objective functions in the scheduling area. We thus have a rich body of such results as of today for many kinds of settings.

An important member of this class is a list scheduling algorithm that is known as LPT (longest processing time first). We assume that we know the processing duration (execution time) of each job. We order them in descending order of processing times. In this case, we can prove that the ratio is bounded by $\left(\frac{4}{3} - \frac{1}{3m}\right)$.

5.4.4 Banker's Algorithm

Let us now look at scheduling with deadlock avoidance. This basically means that before acquiring a lock, we would like to check if a potential lock acquisition may lead to a deadlock or not. If there is a possibility of a deadlock, then we would like to back off. We have already seen a simpler version of this when we discussed the lockdep map in Section 5.3.4. The Banker's algorithm, which we will introduce in this section, uses a more generalized form of the lockdep map algorithm where we can have multiple copies of each resource. It is a very classical algorithm in this space, and is easily implementable in the real-world.

The key insight is as follows. Finding circular waits in a graph is sufficient for cases where we have a single copy of a resource, however, when we have multiple copies of a resource, a circular wait is not well-defined. Refer to Figure 5.29. We show a circular dependency across processes and resources. However, because of multiple copies, a deadlock does not happen. Hence, the logic for detecting deadlocks when we have multiple copies of resources is not as simple as finding cycles in a graph. We need a different algorithm.

Resource	Count
A	2
B	2

1. P1 releases A and B
2. P2 and P3 get them
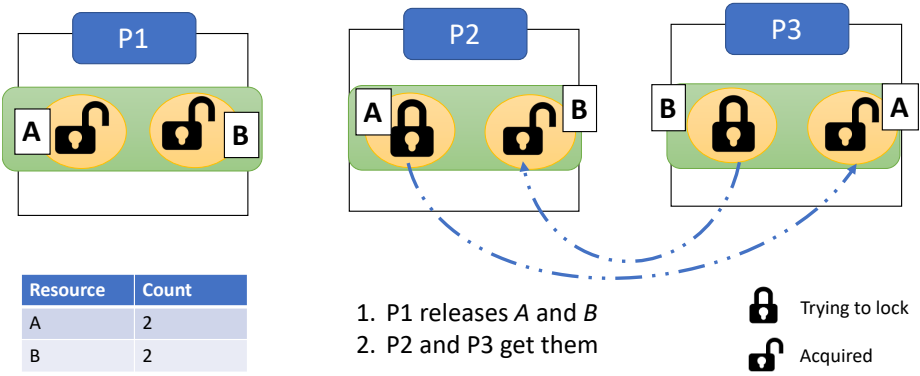
Figure 5.29: A circular dependency between processes $P2$ and $P3$. There are two resources: A and B (two copies each). $P1$ has locked (acquired) a unit of resource A and a unit of resource B. $P2$ has locked a unit of resource B and waits for a unit of resource A. $P3$ has locked a unit of resource A and waits for a unit of resource B. There is a circular dependency between $P2$ and $P3$. $P1$ will ultimately release its resources and the deadlock between $P2$ and $P3$ will break. Sufficient resources will be available. Due to the presence of multiple copies, there is thus no deadlock situation. Every process ultimately gets the resources that it needs.

Data Structures

Let us look at the data structures used in the Banker's algorithm (see Table 5.2). There are n processes and m types of resources. The array `avlbl` stores the number of copies that we are currently available for resource i.

Data structures and their dimensions	Explanation
`avlbl[m]`	`avlbl[i]` stores the number of free copies of resource i
`max[n][m]`	Process i can request at most `max[i][j]` copies of resource j
`acq[n][m]`	Process i has currently acquired `acq[i][j]` copies of resource j
`need[n][m]`	Process i may request `need[i][j]` copies of resource j in the future (at max.). `acq + need = max`

Table 5.2: Data structures in the Banker's algorithm

Safety of States

The algorithm to find if a state is `safe` or not in the Banker's algorithm is shown in Algorithm 1. The basic philosophy is as follows. Given the requirements of all the processes in terms of resources, we do a short hypothetical simulation and see if we can find a schedule to satisfy the requests of all the processes. If this appears to be possible, then we say that the system is in a `safe` state. Otherwise, if we find that we are in a position where the requests of any subset of processes cannot be satisfied, the state is said to be `unsafe`. There is a need to wait till the state becomes `safe` again.

In Algorithm 1, we first initialize the `cur_cnt` array and set it equal to `avlbl` (counts of free resources). At the beginning, the request of no process is assumed to be satisfied (serviced). Hence, we set the value of all the entries in the array `done` to **false**.

Next, we need to find a process with id i such that it is not done yet (`done[i] ==` **false**) and its maximum requirements stored in the `need[i]` array are element-wise less than `cur_cnt`. Let us define some terminology here before proceeding forward. `need[][]` is a 2-D array. `need[i]` is a 1-D array that captures the resource requirements for process i – it is the ith row in `need[n][m]` (row-column format). For two 1-D arrays A and B of the same size, the expression $A \prec B$ means that $\forall i, A[i] \leq B[i]$ and $\exists j, A[j] < B[j]$. This means that each element of A is less than or equal to the corresponding element of B. Furthermore, there is at least one entry in A that is strictly less than the corresponding entry in B. If both the arrays are element-wise identical, we write $A = B$. Now, if either of the cases is true – $A \prec B$ or $A = B$ – we write $A \preceq B$.

Let us now come back to the expression `need[i]` $\preceq$ `cur_cnt`. It means that the maximum requirement of a process is less than or equal to the currently available set of resources (for all resource types). In other words, the request of process i can be satisfied.

Algorithm 1 Algorithm to check for the safety of the system

```
 1: initialize:
 2: cur_cnt ← avlbl
 3: ∀i, done[i] ← false
 4:
 5: find:
 6: if ∃i, (done[i] == false) && (need[i] ⪯ cur_cnt) then
 7:     go to update
 8: else
 9:     go to safety_check
10: end if
11:
12: update:
13: ▷ Release all acquired resources because the request is assumed to be served
14: cur_cnt ← cur_cnt + acq[i]
15: done[i] ← true
16: go to find
17:
18: safety_check:
19: if ∀i, done[i] == true then
20:     return safe
21: else
22:     return unsafe
23: end if
```

If no such process is found, we jump to the last step. It is the safety check step. However, if we are able to find such a process with id i, then we assume that it will be able to execute because enough resources are available. After hypothetical execution, it will return all the resources that it currently holds (i.e., `acq[i]`) back to the free pool of resources (`cur_cnt`). Given that we were able to satisfy the request for process i, we set `done[i]` equal to **true**. Its resources are returned to the free pool of resources and thus `acq[i]` is added to `cur_cnt`. We continue repeating this process till we can satisfy as many requests of processes as we can.

Let us now come to the last step, where we perform the safety check. If the requests of all the processes are satisfied, then all the entries in the `done` array will be equal to **true**. It means that we are in a **safe** state – the requests of all the processes can be satisfied. Alternatively, all the requests that are currently pending can be safely accommodated. Otherwise, we are in an **unsafe** state. It basically means that we have more requirements as compared to the number of free resources. This situation may induce a potential deadlock.

Point 5.4.3

Significance of the **safe** state: The **safe** state means that if every process were to immediately request for the maximum number of resources that it is entitled to request, then it will be possible to satisfy all of them. However, if such a sequence cannot be found out, then the state is **unsafe**. There is another way to understand the **unsafe** state. If every process's

request is equal to its need, then we have a deadlock. This is because the need of at least one process cannot be satisfied. Assuming that the request is equal to the need, is a conservative worst case. However, if it does indicate that the largest possible requests can be accommodated. Note that the initial state when no request has been allocated should ideally be a safe state. This means that it should be possible to satisfy all requests. All of them should be less than the need. This is a sanity check condition. However, once resources have been allocated, the system may move towards an unsafe state. An unsafe state can lead to a deadlock if requests are large enough.

Requesting for Resources

Algorithm 2 Algorithm to request for resources

 1: <u>initialize:</u>
 2: initialize the `req[]` array
 3:
 4: <u>check:</u>
 5: **if** $\neg$ (`req`$_i$ $\preceq$ need[i]) **then**
 6:     **return** false
 7: **else**
 8:     **if** `avlbl` $\prec$ `req` **then**
 9:         `wait()`
10:     **end if**
11: **end if**
12:
13: <u>allocate:</u>
14: `avlbl` $\leftarrow$ `avlbl` - `req`$_i$
15: `acq[i]` $\leftarrow$ `acq[i]` + `req`$_i$
16: `need[i]` $\leftarrow$ `need[i]` - `req`$_i$
17: **if** state is unsafe **then**
18:     Disallow the request and undo changes
19: **else**
20:     **return**
21: **end if**

Let us now look at the resource request algorithm (Algorithm 2). We start out with introducing a new array called `req`$_i$, which holds process i's requirements. For example, if `req`$_i$[j] is equal to k, it means that process i needs k copies of resource j. We need to check if this is a valid request or not. This is a key part of the deadlock avoidance algorithm. If we can pre-certify every request, then we will never enter a deadlocked state.

Consider the <u>check</u> phase. Ideally `req`$_i$ $\preceq$ `need[i]`, which basically means that a process is requesting for resources that it is entitled to. If this is not the case, then at least one entry in `req`$_i$ is strictly greater than the corresponding entry in `need[i]`. The current requirement is clearly more than what was declared a priori (stored in the `need[i]` array). Such requests cannot be satisfied.

Therefore, we need to return **false**.

Assume this is not the case. If `avlbl ≺ req`, then it means that we need to wait for resource availability, which may happen in the future. In this case, we are clearly not exceeding pre-declared thresholds, as we were doing in the former case.

Finally, assume that we have enough available resources. The key question is whether we should allocate the requested resources or not. The answer depends on whether we arrive at a safe state or not. To check this, let us make a dummy allocation (<u>allocate</u> step). The first step is to subtract `req_i` from `avlbl`. This basically means that we satisfy the request for process i (hypothetically). The resources that it requires are not free anymore. This fact is represented by adding `req_i` to `acq[i]`, which basically means that the said resources have been acquired. We then proceed to subtract `req_i` from `need[i]`. This is because at all points of time, the following invariant needs to hold: `max = acq + need`.

Example 5.4.1

Consider a system with two processes P_1 and P_2. The sizes of the arrays are as follows.

$$avlbl = \begin{bmatrix} 1 & 1 \end{bmatrix} \quad acq = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad need = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Assume that P_1 puts forth a request $\begin{bmatrix} 1 & 0 \end{bmatrix}$. The state becomes:

$$acq = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

This state is **safe** because maximum-sized requests of P_1 and P_2 can be satisfied. Now, assume that P_2 issues a request $\begin{bmatrix} 0 & 1 \end{bmatrix}$. If this request were to be granted then, the state will become

$$avlbl = \begin{bmatrix} 0 & 0 \end{bmatrix} \quad acq = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

This is an **unsafe** state. No resource is available and the needs of both the processes are not satisfied. Hence, the request $\begin{bmatrix} 0 & 1 \end{bmatrix}$ should be denied because it leads to an **unsafe** state.

After this dummy allocation, we check if the state is **safe** or not by invoking the algorithm to check for state safety (Algorithm 1). If the state is not **safe**, then it means that the current resource allocation request should not be allowed – it may lead to a deadlock.

Algorithm for Finding Deadlocks

Let us now look at the deadlock detection algorithm (Algorithm 3). We introduce a new `reqs[n][m]` array that stores resource requests for all the processes. For example, `reqs[i]` stores all the resource requests for process i. It is equivalent to `req_i`. We start with the same sequence of initialization steps. We first set `cur_cnt` to `avlbl`. Next, for all the processes that have not acquired any resource, we set `done[i]` to **false**– they cannot be part of a deadlock because

---

Algorithm 3 Algorithm for finding deadlocks

 1: <u>initialize:</u>
 2: cur_cnt ← avlbl
 3: **for** $\forall i$ **do**
 4:     **if** acq[i] $\neq 0$ **then**
 5:         done[i] ← **false**
 6:     **else**
 7:         done[i] ← **true**
 8:     **end if**
 9: **end for**
10:
11: <u>find:</u>
12: **if** $\exists i$, (done[i] == **false**) && reqs[i] $\preceq$ cur_cnt **then**
13:     go to <u>update</u>
14: **else**
15:     go to <u>deadlock_check</u>
16: **end if**
17:
18: <u>update:</u>
19: cur_cnt ← cur_cnt + acq[i]
20: done[i] ← **true**
21: go to <u>find</u>
22:
23: <u>deadlock_check:</u>
24: **if** $\forall i$, done[i] == **true then**
25:     **return** No Deadlock
26: **else**
27:     **return** Deadlock
28: **end if**

---

the hold-and-wait condition is not valid for them. For the rest of the processes, we set `done[i]` to **true**.

> **Point 5.4.4**
>
> Algorithms 3 and 1 are quite similar. The deadlock detection algorithm (Algorithm 3) considers a realistic case, where the requests are pre-specified in the `reqs` array. It detects if the current state has a deadlock or not. Algorithm 1 uses the same logic; however, it assumes the worst case. It assumes that the largest possible set of resources is requested (equal to `need`).

Next, we find an i such that it is not done yet and $\texttt{reqs}[i] \preceq \texttt{cur\_cnt}$. Note that the major difference between this algorithm and the algorithm to check whether a state is safe or not (Algorithm 1) is the use of the `reqs` array here, as opposed to the `need` array in Algorithm 1. `need` represents the maximum number of additional requests a process can request for. Whereas, `reqs` points to the current set of requests. We have the following relationship between them: $\texttt{reqs} \preceq \texttt{need}$.

Let us now understand the expression $\texttt{reqs}[i] \preceq \texttt{cur\_cnt}$. This basically means that for some process i, we can satisfy its request at the current point of time. We subsequently move to update, where we assume that i's request has been satisfied (albeit hypothetically). Therefore, similar to the safety checking algorithm, we return the resources that i has acquired. We thus add `acq[i]` to `cur_cnt`. The process is marked as done ($\texttt{done}[i] \leftarrow$ **true**). We go back to the find step and keep iterating till we can satisfy the requests of as many processes as possible. When this is not possible anymore, we jump to deadlock_check.

Now, if $\texttt{done}[i] ==$ **true** for all processes, then it means that we were able to satisfy the requests of all the processes. Therefore, there cannot be a deadlock. However, if this is not the case, then it means that there is a dependency between processes because of the resources that they are holding. This indicates a potential deadlock situation.

There are several ways of avoiding a deadlock if this algorithm indicates that a deadlock may form. The first is that before every resource/lock acquisition we check the request using Algorithm 2. We do not acquire the resource if we are entering an **unsafe** state. If the algorithm is more optimistic, and we have entered an **unsafe** state already, then we perform a deadlock check, especially when the system does not appear to make any progress. We kill one of the processes involved in the deadlock and release its resources. We can choose one of the processes that has been waiting for a long time or has a very low priority.

5.4.5   Scheduling in the Linux Kernel

The entry point to the scheduling subsystem in the Linux kernel is the `schedule` function (refer to Listing 5.34).

The first task is to finish pending work for the current process. It is possible that the current process is going to sleep. In this case, if there is some pending work and that needs to be assigned to worker threads or some I/O tasks need to be created, then this is the time to do so. The kernel has a set of threads known as *kworker* threads. They perform the bulk of the kernel's low-priority work. This function wakes up worker threads and assigns them some work. This

is especially important if the current task is going to get blocked. It will also creates work items to finish all pending I/O.

Next, the internal version of the __schedule function is called within a code region where preemption is disabled. This allows the __schedule function to execute unhindered and also avoids a bunch of correctness problems associated with accessing concurrent data structures. After the __schedule function returns, we update the status of worker threads (sched_update_worker). This function performs simple bookkeeping. For example, if a worker thread is selected to run on the CPU, then this function helps the scheduler account for the fact that it is doing work related to workqueues. Its fairness policies use this information.

Listing 5.34: The schedule function
source : kernel/sched/core.c#L6681
Also refer to [de Olivera, 2018]

```c
void schedule(void)
{
    struct task_struct *tsk = current;

    /* Dispatch work to other kernel threads */
    sched_submit_work(tsk);

    do {
        preempt_disable();
        __schedule(SM_NONE);  /* scheduling work */
        sched_preempt_enable_no_resched();
    } while (need_resched());

    /* Update the status of worker threads */
    sched_update_worker(tsk);
}
```

There are several ways in which the schedule function can be called. If a task makes a blocking call to a mutex or semaphore, then there is a possibility that it may not acquire the mutex/semaphore. In this case, the task needs to be put to sleep. The state will be set to either INTERRUPTIBLE or UNINTERRUPTIBLE. Since the current task is going to sleep, there is a need to invoke the schedule function such that another task can execute.

The second case is when a process returns after processing an interrupt or system call. The kernel checks the TIF_NEED_RESCHED flag. If it is set to true, then it means that there are waiting tasks and there is a need to schedule them. Similarly, if there is a timer interrupt, there may be a need to swap the current task out and bring a new task in (preemption). Again we need to call the schedule to pick a new task to execute on the current core. It is important to note that the scheduler is not called on every timer interrupt. We shall see later that it is called only when the current task has exceeded its execution time quota.

Every CPU has a runqueue where tasks are added. This is the main data structure that manages all the tasks that are supposed to run on a CPU. The apex data structure here is the runqueue (struct rq) (see kernel/sched/sched.h).

Linux defines different kinds of schedulers (refer to Table 5.3). Each scheduler uses a different algorithm to pick the next task that needs to run on a

core. The internal `__schedule` function is a wrapper function on the individual scheduler-specific function. There are many types of runqueues – one for each type of scheduler.

Scheduling Classes

Let us introduce the notion of scheduling classes. A scheduling class represents a scheduler that manages its own set of jobs and picks the most eligible job for subsequent execution on a CPU. Linux defines a strict hierarchy of scheduling classes. This means that if there is a pending job in a higher scheduling class, then we schedule it first before scheduling a job in a lower scheduling class.

The classes are as follows in descending order of priority.

Stop Task This is the highest priority task. It stops everything and executes. For example, if there is a kernel panic or a new CPU is added, then tasks in this scheduling class are run. This facility should only be used in emergency situations.

DL This is the deadline scheduling class that is used for real-time tasks. Every task is associated with a deadline. Typically, audio and video encoders create tasks in this class. This is because they need to finish their work in a bounded amount of time. For example, for 60-Hz video, the deadline is 16.66 ms.

RT These are regular real-time threads that are used for a host of tasks. There are two ways to schedule such threads: FIFO (first-in-first-out) and RR (round-robin).

Fair This is the default scheduler that the current version of the kernel (v6.2) uses for regular tasks. It ensures a high degree of fairness among tasks where even the lowest priority task gets some CPU time.

Idle This scheduler runs the idle process, which runs when there is nothing to run. It simply puts the CPU into an idle low-power state and sleeps. Having an idle process is a good idea for bookkeeping and also makes the design of the scheduler simple. There is always one task that can be preempted.

There is clearly no fairness across classes. This means that it is possible for *DL* tasks to completely monopolize the CPU and even stop real-time tasks from running. Then the system will become unstable and will crash. It is up to the user to ensure that this does not happen. This means that sufficient computational bandwidth needs to be kept free for regular and real-time tasks such that they can execute. The same holds for real-time tasks as well – they should not monopolize the CPU resources. There is some degree of fairness within a class; however, it is the job of the user to ensure that there is a notion of fairness across classes (system-wide).

Each of these schedulers is defined by a `struct sched_class` object. This is a generic structure that simply defines a set of function pointers (see Listing 5.35). Each scheduler defines its own functions and initializes a structure of type `struct sched_class`. Each function pointer in `sched_class` is assigned

a pointer to a function that implements some functionality of the scheduling class.

This is the closest that we can get to a truly object-oriented implementation. Given that we are not using an object-oriented language in the kernel and using C instead of C++, we do not have access to classical object-oriented primitives such as inheritance and polymorphism. Hence, we need to create something in C that mimics the same behavior. This is achieved by defining a generic `sched_class` structure that contains a set of function pointers. The function pointers point to relevant functions defined by the specific scheduler. They can also be changed at runtime.

For example, if we are implementing a deadline scheduler, then all the functions shown in Listing 5.35 point to the corresponding functions defined for the deadline scheduler. This is a flexible mechanism. We can create a bespoke scheduler that uses a different algorithm.

Listing 5.35: List of important functions in `struct sched_class`
source : kernel/sched/sched.h#L2170

```
/* Enqueue and dequeue in the runqueue */
void (*enqueue_task) (struct rq *rq, struct task_struct *p,
    int flags);
void (*dequeue_task) (struct rq *rq, struct task_struct *p,
    int flags);

/* Key scheduling function */
struct task_struct * (*pick_task)(struct rq *rq);
struct task_struct * (*pick_next_task)(struct rq *rq);

/* Migrate the task and update the current task */
void (*migrate_task_rq)(struct task_struct *p, int new_cpu);
void (*update_curr)(struct rq *rq);
```

In Listing 5.35, we observe that most of the functions have the same broad pattern. The key argument is the runqueue `struct rq` that is associated with each CPU. It contains all the `task_struct`s scheduled to run on a given CPU. In any scheduling operation, it is mandatory to provide a pointer to the runqueue such that the scheduler can find a task among all the tasks in the runqueue to execute on the core. We can additionally perform several operations on the runqueue such as enqueueing or dequeueing a task – `enqueue_task` and `dequeue_task` – respectively.

The most important functions in any scheduler are the functions `pick_task` and `pick_next_task` – they select the next task to execute. These functions are scheduler specific. Each type of scheduler maintains its own data structures and has its own internal notion of priorities and fairness.

The `pick_task` function is the fast path that finds the highest priority task (all tasks are assumed to be independent), whereas the `pick_next_task` function is on the slow path. The slow path incorporates some additional functionality, which can be explained as follows. Linux has the notion of control groups (*cgroups*). These are groups of processes that share scheduling resources. Linux ensures fairness across processes and cgroups. In addition, it ensures fairness across processes in a cgroup. cgroups further can be grouped into hierarchies. A cgroup can act as a parent and have several child cgroups as its children.

The `pick_next_task` function ensures fairness while also considering cgroup information.

Let us consider a few more important functions. `migrate_task_rq` *migrates* the task to another CPU; it performs the crucial job of load balancing. `update_curr` performs some bookkeeping for the current task; it updates its runtime statistics. There are many other functions in this class such as functions to yield the CPU, check for preemptibility, set CPU affinities and change priorities.

These scheduling classes are defined in the kernel/sched directory. Each scheduling class has an associated scheduler, which is defined in a separate C file (see Table 5.3).

Scheduler	File
Stop task scheduler	stop_task.c
Deadline scheduler	deadline.c
Real-time scheduler	rt.c
Completely fair scheduler (CFS)	cfs.c
Idle	idle.c

Table 5.3: List of schedulers in Linux

The runqueue

Let us now take a deeper look at a runqueue (`struct rq`) in Listing 5.36. The entire runqueue is protected by a single spinlock `__lock`. It is used to lock all key operations on the runqueue. Such a global lock that protects all the operations on a data structure is known as a *monitor lock*.

The next few fields are basic CPU statistics. The field `nr_running` is the number of runnable processes in the runqueue. `nr_switches` is the number of process switches recorded on the CPU and the field `cpu` is the CPU number.

The runqueue is actually a container of individual scheduler-specific runqueues. It contains three fields that point to runqueues of different schedulers: `cfs`, `rt` and `dl`. They correspond to the runqueues for the CFS, real-time and deadline schedulers, respectively. We assume that in any system, at the minimum we will have three kinds of tasks: regular tasks (handled by CFS), real-time tasks and tasks that have a deadline associated with them. These scheduler types are hardwired into the logic of the runqueue.

It holds pointers to the current task (`curr`), the idle task (`idle`) and the `mm_struct` of the last user process that ran on the CPU (`prev_mm`). The task that is chosen to execute is stored in `task_struct *core_pick`.

Listing 5.36: The runqueue
source : kernel/sched/sched.h#L954

```
struct rq {
    /* Lock protecting all operations */
    raw_spinlock_t      __lock;

    /* Basic CPU stats */
    unsigned int        nr_running;
```

```
    u64                 nr_switches;
    int                 cpu;

    /* One runqueue for each scheduling class */
    struct cfs_rq       cfs;
    struct rt_rq        rt;
    struct dl_rq        dl;

    /* Pointers to the current task, idle task and the mm\
        _struct */
    struct task_struct    *curr;
    struct task_struct    *idle;
    struct mm_struct      *prev_mm;

    /* The task selected for running */
    struct task_struct *core_pick;
};
```

Picking the Next Task

Let us consider the slow path. The `schedule` function calls `pick_next_task`, which invokes the internal function `__pick_next_task`. As we have discussed, this is a standard pattern in the Linux kernel. The functions starting with "`__`" are functions internal to a file.

We iterate through the classes and choose the highest priority class that has a queued job. We run the scheduling algorithm for the corresponding `sched_class` and find the task that needs to execute. In the case of regular processes, we run the CFS scheduler to pick the next task. Subsequently, we effect a context switch.

Scheduling algorithms typically have a similar structure. They heavily rely on the execution statistics of tasks. For example, they may rely on the duration of the current task's execution, number of migrations and context switches recorded in the recent past. This information is used to achieve a balance between high performance and fairness. Let us focus on the information that is used by popular Linux schedulers.

Scheduling-Related Statistics and Metadata

Listing 5.37: Scheduling-related fields in the `task_struct`
source : include/linux/sched.h#L788

```
/* Scheduling statistics */
struct sched_entity       se;
struct sched_rt_entity    rt;
struct sched_dl_entity    dl;
const struct sched_class    *sched_class;

/* Preferred CPU */
struct rb_node              core_node;

/* Identifies a set of group of tasks that can safely
    execute on
```

```
the same core. This is from a security standpoint */
unsigned long                    core_cookie;
```

Let us now look at some scheduling-related statistics (see Listing 5.37) stored in the `task_struct` structure. There are different types of `sched_entity` classes (one for each scheduler type) that store this information. For example, for CFS scheduling, the `sched_entity` structure contains relevant information such as the time at which execution started, cumulative execution time, the virtual runtime (relevant to CFS scheduling), the number of migrations, etc. Based on the scheduling class (field: `sched_class`), the appropriate `sched_entity` is chosen. Clearly, different scheduling classes rely on different kinds of information. Let us now look at some important pieces of metadata.

The `core_node` and `core_cookie` fields in `task_struct` are important fields that have crucial performance and security implications. Specifically, `core_node` points to the core that is the "home core" of the process. This means that by default the process is scheduled on the core associated with `core_node`, and all of its memory is allocated in close proximity to `core_node`. This is particularly important on a NUMA (non-uniform memory access) machine, where the notion of the proximity of memory modules to a node exists.

The `core_cookie` uniquely identifies a set of tasks. All of them can be scheduled on the same core (or group of cores) one after the other. They are deemed to be *mutually safe*. This means that they are somehow related to each other and are not guaranteed to mount attacks each other. If we schedule unrelated tasks on the same core, then it is possible that one task may use architectural side-channels to steal secrets from another task running on the same core. Hence, there is a need to restrict this set using the notion of a core cookie.

5.4.6 Completely Fair Scheduling (CFS)

The CFS scheduler is the default scheduler for regular processes (in kernel v6.2). It ensures that every process gets at least one execution time quantum in a scheduling period. There is no starvation. As discussed earlier, the `sched_entity` class maintains all scheduling-related information for CFS tasks (refer to Listing 5.38). Let us now look at it in some more detail.

struct sched_entity

Listing 5.38: `struct sched_entity`
source : include/linux/sched.h#L547

```
struct sched_entity {
    struct load_weight  load; /* for load balancing*/
    struct rb_node      run_node;

    /* statistics */
    u64      exec_start;
    u64      sum_exec_runtime;
    u64      vruntime;
    u64      prev_sum_exec_runtime;
    u64      nr_migrations;
```

```
    struct sched_avg   avg;

    /* runqueue */
    struct cfs_rq      *cfs_rq;
};
```

The CFS scheduler manages multiple cores. It maintains per-CPU data structures and tracks the load on each CPU. It migrates tasks as and when there is a large imbalance between CPUs.

Let us now focus on the per-CPU information that it stores. The tasks in a CFS runqueue are arranged as a red-black (RB) tree sorted by their vruntime (corresponding to a CPU). The virtual runtime (*vruntime*) is the key innovation in CFS schedulers. It can be explained as follows. For every process, we keep a count of the amount of time that it has executed for. Let's say a high-priority task executes for 10 ms. Then instead of counting 10 ms, we actually count 5 ms. 10 ms in this case is the actual runtime and 5 ms is the virtual runtime (vruntime). Similarly, if a low-priority process executes for 10 ms, we count 20 ms – its vruntime is larger than its actual execution time. Now, when we use vruntime as a basis for comparison and choose the task with the lowest vruntime, then it is obvious that high-priority tasks get a better deal. Their vruntime increases more sluggishly than low-priority tasks. This is in line with our original intention where our aim was to give more CPU time to higher priority tasks. We arrange all the tasks in a red-black tree. Similar to the way that we add nodes in a linked list by having a member of type **struct list_head**, we do the same here. To make a node a part of a red-black tree, we include a member of type **struct rb_node** in it. The mechanism of accessing the red-black tree and getting a pointer to the encapsulating structure is the same as that used in lists and hlists.

The rest of the fields in **struct sched_entity** contain various types of runtime statistics such as the total execution time, total vruntime, last time the task was executed, etc. It also contains a pointer to the encapsulating CFS runqueue.

Notion of vruntimes

Listing 5.39: weight as a function of the nice value
source : kernel/sched/core.c#L11338

```
const int sched_prio_to_weight[40] = {
/* -20 */ 88761, 71755, 56483, 46273, 36291,
/* -15 */ 29154, 23254, 18705, 14949, 11916,
/* -10 */ 9548,  7620,  6100,  4904,  3906,
/* -5 */  3121,  2501,  1991,  1586,  1277,
/*  0 */  1024,  820,   655,   526,   423,
/*  5 */  335,   272,   215,   172,   137,
/* 10 */  110,   87,    70,    56,    45,
/* 15 */  36,    29,    23,    18,    15,
};
```

The increment in vruntime $\delta_{vruntime}$ is proportional to the actual runtime δ of the last interval. The relationship is shown in Equation 5.4. The scaling factor is equal to the weight associated with the *nice* value of 0 divided by the

weight associated with the real nice value. Refer to Section 3.1.7, where we discussed the meaning of nice values. In this case, we clearly expect the ratio to be less than 1 for high-priority tasks and be more than 1 for low-priority tasks.

$$\delta_{vruntime} = \delta \times \frac{weight(nice = 0)}{weight(nice)} \tag{5.4}$$

Listing 5.39 shows the mapping between nice values and weights. The nice value is 1024 for the nice value 0, which is the default. For every increase in the nice value by 1, the weight reduces by roughly $1.25\times$. For example, if the nice value is 5, the weight is 335. $\delta_{vruntime} = 3.06\delta$. Clearly, we have an exponential decrease in the weight as we increase the nice value. For a nice value of n, the weight is roughly $1024/(1.25)^n$. The highest priority user task has a weight equal to 88761 ($86.7\times$). This means that it gets significantly more runtime as compared to a task that has the default priority.

Scheduling Periods and Slices

However, this is not enough. We need to ensure that in a scheduling period (long duration of time), every task gets at least one chance to execute. This is more or less ensured with virtual runtimes. However, there is a need to ensure this directly and provide a stricter notion of fairness. We wish to give every process a chance to execute at least once in a scheduling period.

Before we proceed further, let us provide some background. Consider the following variables.

`sysctl_sched_latency` (SP)	The scheduling period in which all tasks run at least once.
`sched_nr_latency` (N)	Maximum number of runnable tasks that can be considered in a scheduling period for execution.
`sysctl_sched_min_granularity` (G)	Minimum amount of time that a task runs in a scheduling period

Let us use the three mnemonics SP, N and G for the sake of readability. Refer to the code snippet shown in Listing 5.40. If the number of runnable tasks is more than N (limit on the number of runnable tasks that can be considered in a scheduling period SP), then it means that the system is swamped with tasks. We clearly have more tasks than what we can run. This is a crisis situation, and we are looking at a rather unlikely situation. The only option in this case is to increase the scheduling period by multiplying `nr_running` with G (minimum task execution time).

Let us consider the *else* part, which is the more likely case. In this case, we set the scheduling period as SP.

Listing 5.40: Implementation of scheduling quanta in CFS
source : kernel/sched/fair.c#L725

```
u64 __sched_period(unsigned long nr_running)
{
    if (unlikely(nr_running > sched_nr_latency))
        return nr_running * sysctl_sched_min_granularity;
    else
```

```
        return sysctl_sched_latency;
}
```

Once the scheduling period has been set, we set the *scheduling slice* for each task as shown in Equation 5.5 (assuming we have the normal case where `nr_running` $\leq$ `N`).

$$slice_i = SP \times \frac{weight(task_i)}{\sum_j weight(task_j)} \qquad (5.5)$$

We basically partition the scheduling period based on the weights of the constituent tasks. Clearly, high-priority tasks get larger scheduling slices. However, if we have the highly unlikely case where `nr_running` > `N`, then each slice is equal to G.

CFS Scheduling

The scheduling algorithm works as follows. We find the task with the least vruntime in the red-black tree. We allow it to run until it exhausts its scheduling slice. This logic for this part is shown in Listing 5.41. At this point, if the CFS queue has more than one runnable task, then there may be a need for scheduling. We compute the time for which the current task has already executed and store the result in `ran`. If `slice` > `ran`, then we execute the task for `delta = slice - ran` time units by setting a high-resolution timer accordingly. Otherwise, there is a need to invoke the scheduler. This is because the current task has exhausted its allocated scheduling slice.

Listing 5.41: `hrtick_start_fair`
source : kernel/sched/fair.c#L6012

```
if (rq->cfs.h_nr_running > 1) {
    /* There is more than one task */

    /* Fetch the size of the slice */
    u64 slice = sched_slice(cfs_rq, se);
    u64 ran = se->sum_exec_runtime - se->
        prev_sum_exec_runtime;

    s64 delta = slice - ran;      /* time left */

    if (delta < 0) {          /* exhausted its time slice */
        if (task_current(rq, p))
            resched_curr(rq);      /* invoke the scheduler */
        return;
    }

    /* Set an alarm and execute the rest of the slice */
    hrtick_start(rq, delta);
}
```

Equivalence to Round-Robin Execution

Let us now look at the insights behind the design of the CFS scheduler. Let us start with a simple example shown in Figure 5.30.
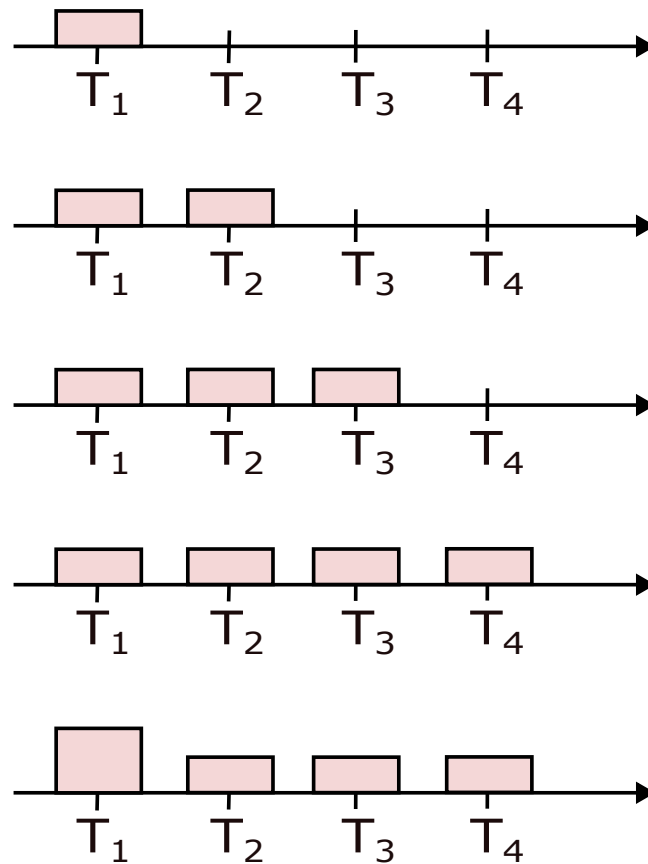
Figure 5.30: Effective round-robin execution in CFS scheduling

Consider a system with four tasks: T_1, T_2, T_3 and T_4. Initially, their cumulative execution time is zero. Assume that they have the same priority. If they are run using the CFS scheduler, then T_1 will run for the duration of its scheduling slice. After that T_2 will run, so on and so forth. Once all of them complete their slices, T_1 will run again and complete its second slice. It is very easy to observe that we are following a round-robin model of execution, which is a fair execution if all the tasks have the same priorities.

Now, consider the case when the priorities of the tasks are different. They will not have such an execution pattern. It will be quite different. The reason is that at every point, we choose the task that has the least vruntime. Let us now prove that in terms of vruntime, the CFS schedule is a round-robin schedule. Consider the normal case, when we are not overwhelmed with tasks.

We have the following relationships between different variables for task T. Assume that the actual time of execution δ is equal to the slice. This means that the task completes its slice.

$$slice \propto weight(T) \quad \text{(by Eqn.5.5)}$$

$$\delta_{vruntime} \propto \frac{slice}{weight(T)} \quad \text{(by Eqn.5.4)} \tag{5.6}$$

$$\Rightarrow \delta_{vruntime} = constant$$

This means that the increase in the vruntime after executing a scheduling slice is a constant, which is not dependent on the priority of the task. This means that if we change the y-axis of Figure 5.30, then we are effectively still doing round-robin scheduling. Instead of using physical time, we are using the vruntime.

Some Important Caveats

Every user has an incentive to increase its execution time by fair or unfair means. They will thus try to game the scheduler. We need to ensure that this does not happen. This means that we need to make special considerations for new tasks, tasks waking up after a long sleep duration and tasks getting migrated from other cores. They will start with a zero vruntime and shall continue to have the minimum vruntime for a long time. This has to be prevented – it is unfair for existing tasks. Also, when tasks move from a heavily-loaded CPU to a lightly-loaded CPU, they should not have an unfair advantage. They will be coming in with a low vruntime. The following safeguards are in place.

1. The `cfs_rq` maintains a variable called `min_vruntime` (lowest vruntime of all processes).

2. Let `se` be the `sched_entity` of the new task that is being restored, migrated or created.

3. If an old task is being restored after a long duration or a new task is being added, then we set $se-> vruntime+ = cfs\_rq-> min\_vruntime$. This ensures that the vruntime of the new task is at least `min_vruntime`. Some degree of a level playing field is being maintained.

4. Moreover, existing tasks have a fair chance of getting scheduled. The vruntime of a new task is at least equal to the minimum and thus it will not continue to enjoy an advantage for a long time.

5. Always ensure that all vruntimes monotonically increase (in the `cfs_rq` and `sched_entity` structures). This is happening by construction and due to the earlier steps.

6. When a child process is created, it inherits the vruntime of the parent. This means that we cannot game the scheduler by creating child processes. No child process enjoys an unfair advantage.

7. Upon a task migration or block/unblock, $cfs\_rq-> vruntime$ is subtracted from the vruntime of the task. This ensures that its vruntime is a relative value with respect to the minimum in the queue. Subsequently, if it is blocked, it continues to retain this value. When it wakes up, its

instantaneous vruntime is this offset added to the minimum vruntime at the time of waking up. If it is migrated to another core, then its vruntime on the destination core is the sum of this offset and `min_vruntime` of the `rq` of the destination core. This strategy ensures that if a task has just finished executing its scheduling slice, it will not get an unfair advantage if it migrates to another core. Otherwise, there is an incentive to continuously keep migrating.

8. Treat a group of processes (in a cgroup) as a single schedulable entity with a single vruntime. This means that processes cannot monopolize CPU time by spawning new processes within a cgroup.

Calculating the CPU Load

Computing the load average on a CPU is important for taking task migration-related decisions. We can also use this information to modulate the CPU's voltage and frequency. The load average needs to give more weightage to *recent activity* as opposed to activity in the past.

We divide the timeline into 1 ms intervals. If a jiffy is 1 ms, then we are essentially breaking the timeline into jiffies. Let such *intervals* be numbered $p_0, p_1, p_2 \ldots$. Let u_i denote the fraction of time in p_i, in which a runnable task executed.

The load average is computed in Equation 5.7.

$$load_{avg} = u_0 + u_1 \times y + u_2 \times y^2 + \ldots \tag{5.7}$$

This is a time-series sun with a decay term y. The decaying rate is quite slow. $y^{32} = 0.5$, or in other words $y = 2^{-\frac{1}{32}}$. This is known as per-entity load tracking (PELT, kernel/sched/pelt.c), where the number of intervals for which we compute the load average is a configurable parameter.

5.4.7 Deadline and Real-Time Scheduling

Deadline Scheduler

The deadline scheduling algorithm is implemented in the `pick_next_task_dl` function. We maintain a red-black tree. The task with the earliest deadline is selected. Recall that the earliest deadline first (EDF) algorithm minimizes the lateness L_{max}. We have similar structures as `sched_entity` here. The analogous structure in every `task_struct` is `sched_dl_entity` – these structures are arranged in ascending order according to their corresponding deadlines.

Real-Time Scheduler

The real-time scheduler has one queue for every real-time priority. In addition, we have a bit vector – one bit for each real-time priority. The bit is set if the corresponding queue has a schedulable job. The scheduler finds the highest-priority non-empty queue. It starts picking tasks from that queue. If there is a single task, then that task executes. Because the scheduler offers no mechanism to balance workloads across real-time priorities, the overall scheme is inherently unfair.

However, for tasks having the same real-time priority, there are two options: FIFO and round-robin (RR). In the real-time FIFO option, we break ties between two equal-priority tasks based on when they arrived (first-in first-out order). In the round-robin (RR) algorithm, we check if a task has exceeded its allocated time slice. If it has, we put it at the end of the queue (associated with the real-time priority). We find the next task in this queue and mark it for execution.

5.5 Real-Time Systems

Up till now we have only looked at regular systems. They have jobs with different priorities. We defined regular user-level priorities and real-time priorities. The latter are for real-time jobs? With real-time priorities, we are guaranteed to schedule real-time jobs before regular priority jobs. This is as far as we can go in a general-purpose Linux system that is not specifically tailored for running mission critical real-time applications. Sadly, regular versions of linux are not fit for use in mission critical systems such as missiles, rockets, defense equipment and medical devices. This is because large parts of the kernel are not preemptible, high-priority tasks can be blocked for a long time by low-priority tasks and the scheduling algorithms do not make strict guarantees regarding adherence to deadlines. Hence, there is a need to specifically look at real-time kernels that are designed ground up. They are specifically designed to execute jobs with real-time constraints, where each job is associated with a deadline.

5.5.1 Types of Real-Time Systems

In this context, we can define three kinds of real-time systems. ① Soft real-time systems are supported by generic versions of Linux. In this case, jobs are associated with a deadline. Such tasks are often periodic and they spawn a separate job instance in each period. They can be scheduled using the real-time or deadline schedulers. With a high probability, we can guarantee adherence to deadlines. Especially with deadline scheduling, tasks can mostly finish before their deadline if the deadline is reasonably chosen (as we shall see later). Note that once in a while, deadlines can be violated even while using such scheduling classes because the basic kernel does not provide stronger guarantees. This can be modeled by a utility function. The user derives a utility, which is a function of the job completion time. In the case of a soft real-time system, the utility is non-zero even when the job overshoots its deadline. Note that the utility in such cases is lower than the utility of jobs that complete in a timely fashion. The utility is typically a decreasing function of time after the deadline is missed. Typically, audio and video processing applications fall in this category. Occasionally missing a deadline will lead to some kind of distortion, but that level of distortion is often acceptable.

② Then we have firm real-time systems. In this case, if the task is delayed beyond its deadline, then it is virtually of no use. In other words, the utility is zero. However, this does not lead to system failure. An example of this is an ATM machine, where if there is a delay, the machine times out and releases the debit card. The utility is zero given that it clearly reduces the perception of the bank in the eyes of the public. The point to note is that it does not

lead to system failure. Another example is a flight reservation system where it is important to coordinate across various systems and finally issue the ticket. These systems include the credit card company, the bank that has issued it, the airline, the travel portal and the client application that has been used to book the ticket. There can be long delays in this process and if the delay exceeds a certain time bound, the ticket is often not booked. In this case, the utility is clearly zero post the deadline, and there are negative consequences in terms of perception. But there is no system failure.

③ A hard real-time system has very strict deadlines. If a task misses its deadline, not only is the utility zero, it also leads to catastrophic consequences that include complete system failure. Mission critical systems such as rockets, for instance, fall in this category. There are other important use cases that can be classified as hard real-time systems such as medical devices like pacemakers, airline collision avoidance systems and nuclear plant control systems.

For both hard and firm real-time systems, we need to redesign operating systems as well as the hardware, so that we can guarantee that the tasks complete within their deadlines. We also need a solid theoretical framework to guarantee that if real-time tasks are compliant with certain constraints, then it is possible to theoretically prove that the set of spawned jobs are schedulable (all their constituent jobs are schedulable). This means it is possible to find a real-time schedule, where no job misses its deadline.

Often in such systems, the set of tasks are known a priori, and they are also very well characterized. For example, most of these systems consider periodic tasks, where the deadline is equal to the period. If the period is let's say 3 seconds, then a new job is created once every 3 seconds, and it is expected to also complete within that time interval, which is basically before the next job in the periodic task is created.

An important problem that is studied in the real-time systems literature is that if a system has a set of periodic tasks, can they be run without missing a deadline? In other words, are a set of periodic tasks schedulable?

5.5.2 EDF Scheduling

Consider the case of uniprocessor scheduling. We have already looked at the Earliest Deadline First [Mall, 2009] or EDF algorithm in Section 5.4.2. The main aim was to minimize L_{max}. L_{max} is defined as the maximum lateness. Recall that lateness was defined as the time between a job's completion time and the deadline. Clearly, we want $L_{max} \leq 0$. This would mean that the set of tasks is schedulable – tasks complete before the deadline.

We had looked at tasks that are aperiodic and can arrive at any point of time in Section 5.4.2. Preemption was enabled. Now, we will look at periodic tasks, where the deadline of any job is equal to its period. This is a different version of the formulation. Here also preemption is enabled.

Let us now look at some theoretical results. Let the duration of a task in a periodic real-time job be d_i and its period be P_i. As mentioned earlier, the deadline is the same as the period. Let us define the utility as follows. The utility is the sum of d_i/P_i for all the periodic real time jobs in the system. It turns out that if the utility $U \leq 1$, then all the periodic jobs are schedulable by EDF, otherwise they are not schedulable using any algorithm. This is arguably one of the most important results in uniprocessor real-time systems. It is used

heavily, and is very simple to implement. All that we need to do is store tasks in a priority queue, and order tasks by their deadline. Finding a feasible schedule is also very easy in this case.

The proof follows the broad approach that was outlined in Section 5.4.2 (for shortest job first scheduling). We use a task-swapping-based argument to show that the EDF algorithm indeed produces feasible schedules if there is one.

> **Point 5.5.1**
>
> If the total utilization ($\sum_i \frac{d_i}{P_i} \leq 1$), it is always possible to find a feasible schedule using the EDF algorithm.

Now, if $U > 1$, it is clear that we need to do more work than what we can do in a given period of time. This is obviously not possible. Hence, if the utilization exceeds 1, a feasible schedule simply cannot be generated by any algorithm. To summarize, the greatness of the EDF algorithm is that it generates a feasible schedule if there is one.

EDF in a certain sense uses dynamic priorities. This means that if a new task arrives when an existing task is executing, the new task can preempt the existing task if its deadline is earlier. This essentially means that the priorities in the system tend to vary over time, i.e., based on the proximity of the deadline. This requires continuous bookkeeping.

5.5.3 RMS Scheduling

Let us consider a static priority system where all the priorities are decided at the beginning. It is assumed that all the periodic tasks that will execute on a system are known a priori. They are very well characterized in terms of their maximum execution time and period. This is almost always the case in hard real-time systems, where the degree of unpredictability is quite low. In such cases, the job priority and the task priority are the same. Note that this was not the case in EDF. If a periodic task spawned multiple jobs, then all of them could have had different priorities. In fact, the priority of a job could have varied across its lifetime.

Liu-Layland Bound

In this context, let us introduce the RMS scheduling algorithm [Mall, 2009]. RMS stands for Rate Monotonic Scheduling, where the priority is inversely proportional to the period of the task. RMS is also a preemptive algorithm. The priorities are known beforehand, and there is no dynamic computation of priorities.

In this case also, the total utilization U of the system is used to determine schedulability. Let n be the number of periodic tasks in the system. It is possible to show that if $U \leq n \left(2^{\frac{1}{n}} - 1\right)$, then the set of jobs is schedulable. This is known as the Liu-Layland bound. When n is equal to 2, $U_{max} = 0.83$ (maximum possible utilization as per the Liu-Layland bound). When $n = \infty$, $U_{max} = 0.69$ (natural logarithm of 2). This bound was originally proposed in 1973. It is quite conservative, and it turns out that we can do better. This is a sufficient condition for schedulability, but it is not necessary. This means that

we can still schedule jobs in some cases when the utilization is greater than the Liu-Layland bound. Let us point out some insights before we move to a less conservative result. It can be shown that the worst case occurs from the point of view of schedulability, when all the tasks start at the same time (are in phase).

> **Definition 5.5.1**
>
> Liu-Layland Bound The Liu-Layland bound states that when $U \leq n\left(2^{\frac{1}{n}} - 1\right)$, the set of periodic tasks are always schedulable.

Lehoczky's Test

The Lehoczky's test [Lehoczky, 1990] proposed a tighter bound. It is also a schedulability test. It can guarantee schedulability for larger values of utilization and includes necessary conditions as well. We check whether every task is meeting its first deadline or not for the worst case (all tasks are in phase, i.e., start at the same time). If this is happening, then we decide that the system is schedulable. Summary: If the utilization exceeds the Liu-Layland bound, the system may still be schedulable as long as it passes the Lehoczky's test.

Let us describe the mathematical details of the Lehoczky's test. We arrange all the periodic jobs in descending order of their RMS priority. Then we compute Equation 5.8.

$$
\begin{aligned}
W_i(t) &= \sum_{j=1}^{i} d_j \left\lceil \frac{t}{P_j} \right\rceil \\
Q_i(t) &= \frac{W_i(t)}{t} \\
Q_i &= \min_{\{0 < t \leq P_i\}} Q_i(t) \\
Q &= \max_{\{1 \leq i \leq n\}} Q_i \\
Q &\leq 1 \qquad \text{All tasks are schedulable}
\end{aligned}
\tag{5.8}
$$

We consider a time interval t, and find the number of periods of task j that are contained within it (fully or partially). Then we multiply the number of such periods with the execution time of each spawned job of task j. This is the total CPU load for task j in the time period t. If we aggregate the loads for the first i tasks in the system (arranged in descending order of RMS priority), we get the cumulative CPU load $W_i(t)$. Let us subsequently compute $Q_i(t) = W_i(t)/t$. It is the mean load of the first i tasks computed over the time interval t.

Next, let us minimize this quantity over a time interval of P_i (period of task i). This means that we find the smallest value of $Q_i(t)$ within this duration. Let this quantity be Q_i, which is equal to $Q_i(t^*)$. If $Q_i \leq 1$, then the ith task is schedulable. It is easy to intuitively see why this is the case. We start at the worst case point (all the tasks are in phase). Subsequently, at t^*, the value of $Q_i(t)$ is minimized. It can be proven that this point determines the schedulability of the ith task (in terms of priority). Hence, we can claim that

the ith task is schedulable if $Q_i(t^*) \leq 1$. If the value of $Q_i(t)$ is always greater than 1 in the time interval P_i, then the ith task is not schedulable.

Next, let us define $Q = max(Q_i)$. If $Q \leq 1$, then it means that $\forall i : Q_i \leq 1$. Hence, all tasks are schedulable. It turns out that this is both a necessary and sufficient condition. For obvious reasons, it is not as elegant and easy to compute as the Liu-Layland bound. Nevertheless, this is a more exact expression and is often used to assess schedules.

5.5.4 DMS Scheduling

A key restriction of RMS is the fact that the deadline needs to be equal to the period. Let us now relax this restriction and allow d_i to be less than P_i. We need to use the Deadline Monotonic Scheduling (DMS) algorithm in this case. Here also we consider a preemptive algorithm with statically defined priorities.

Let the priority of a task be inversely proportional to its deadline (relative to the arrival of the corresponding job). In this case, we need to consider both the duration of a task d_i, and the interference I_i caused by other higher priority tasks. We need to thus ensure that $I_i + d_i \leq D_i$. This will ensure that the task completes before its deadline. However, this is not enough for the entire system. We need to use this basic primitive to design an algorithm that looks at the schedulability of the entire system.

Let us first quantify the *interference* as a function of the interval of time in consideration. It is similar to the expression computed in the Lehoczky's test.

$$I_i(t) = \sum_{j=1}^{i-1} \left\lceil \frac{t}{P_j} \right\rceil \times d_j \tag{5.9}$$

The DMS algorithm is shown in Algorithm 4, which shows the schedulability test for the ith task.

Algorithm 4 The DMS algorithm (schedulability test for the ith task)

1: **for** task τ_i **do**
2:     $\texttt{t} \leftarrow \sum_{j=1}^{i} \texttt{d}_j$
3:     $\texttt{cont} \leftarrow$ **true**
4:     **while** $\texttt{cont}$ **do**
5:         **if** $\texttt{t} > \texttt{D}_i$ **then return false**
6:         **end if**
7:         **if** $\texttt{I}_i(\texttt{t}) + \texttt{d}_i \leq \texttt{t}$ **then**
8:             $\texttt{cont} \leftarrow$ **false**
9:         **else**
10:             $\texttt{t} \leftarrow \texttt{I}_i(\texttt{t}) + \texttt{d}_i$
11:         **end if**
12:     **end while**
13: **end for**

We first compute the value of the variable $\texttt{t}$. It is initialized as the sum of the task execution times of the first i tasks. Recall that these tasks are arranged in descending order of priority. The variable $\texttt{t}$ thus captures the time that is required to execute each of the i tasks once. Next, we initialize $\texttt{cont}$ to **true**, and enter the first iteration of the *while* loop.

Given the value of `t`, we compute the interference using Equation 5.9. It is basically the time that higher priority tasks execute within the first `t` units of time. Next, we compute the sum of the interference and the execution time of task i and check if it is less than or equal to `t`. If this check is successful, it means that the ith task can be scheduled. There is sufficient slack in the overall schedule. We then set `cont` to false. There is no need to keep iterating.

However, if the check fails, it does not mean that the ith task is not schedulable. We need to give it a few additional chances. We consider a longer interval of time and check for schedulability again. Note that we enter the *else* part only when $I_i(t) + d_i > t$. We subsequently set the new value of `t` to be equal to the sum of the interference ($I_i(t)$) and the execution time of the ith task (d_i). We basically set $t \leftarrow I_i(t) + d_i$.

Before proceeding to the next iteration, it is necessary to perform a sanity check. We need to check if `t` exceeds the deadline D_i or not. If it exceeds the deadline, clearly the ith task is not schedulable. We return false. If the deadline has not been exceeded, then we can proceed to the next iteration and repeat the same set of steps.

Given the fact that in every iteration we increase `t`, we will either find task i to be schedulable or `t` will ultimately exceed D_i.

5.5.5 Priority Inheritance Protocol (PIP)

Priority Inversion

Along with scheduling, the other important problem in real-time systems is <u>resource allocation</u>. Some key issues center around deadlocks, fairness and performance guarantees. We have already looked at deadlocks due to improper lock acquisition in Section 5.3.4. We will look at more such issues in the next few sections.

Let us recapitulate. The key idea is that if a low-priority task holds a resource, then it stops a high-priority task that is interested in acquiring the same resource from making any progress. This can in fact lead to a deadlock when both the tasks are confined to the same CPU as we have seen earlier (context inconsistency). The high-priority task will continue to busy-wait. It will not release the CPU for the low-priority task to run. This is a deadlock, which the *lockdep* mechanism tries to detect and subsequently correct.

However, there are clear fairness and performance concerns here as well even if we allow the tasks to run on different CPUs. A low-priority task can *block* a high-priority task. This phenomenon is known as *priority inversion*, which effectively breaks the notion of real-time priorities. Therefore, all real-time operating systems try to avoid such a situation.

> **Definition 5.5.2** Priority Inversion
>
> Priority inversion is a phenomenon where a low-priority task blocks a high-priority task because the former holds a resource that the latter is interested in.

Let us first consider a simple setting where there are two tasks in the system. The low-priority task happens to lock a resource first. When the high-priority task tries to access the resource, it *blocks*. However, in this case, the blocking

time is predictable – it is the time that the low-priority task will take to finish using the resource. After that the high-priority task is guaranteed to run. This represents a simple case and is an example of *bounded priority inversion.*

Let us next consider a more complicated case. Assume that a high-priority task is blocked by a low-priority task. This low-priority task got preempted by a medium priority task. This medium-priority task has ended up blocking the high-priority task unbeknownst to it. If such medium-priority tasks continue to run, the low-priority task may remain blocked for a very long time. Here, the biggest loser is the high-priority task because the time for which it will remain blocked is not known and is dependent on the behavior of many other tasks. Hence, this scenario is known as *unbounded priority inversion.*

Next, assume that a task needs access to k resources, which it needs to acquire sequentially (one after the other). It may undergo priority inversion (bounded or unbounded) while trying to acquire each of these k resources. The total amount of time that the high-priority task spends in the *blocked* state may be prohibitive. This situation needs to be prevented. Assume that these are nested locks – the task acquires resources without releasing previously held resources. A crucial question we need to answer when we introduce our protocols is whether after acquiring a resource, a task gets blocked while acquiring subsequent resources. If it gets blocked, we shall refer to this phenomenon as *second blocking.*

We can also have chain blocking, where a task T_1 is blocked by another task T_2, and T_2 is blocked by another task T_3. Here, we are assuming that the reason for blocking is a resource conflict. For example, T_1 needs to lock a resource that is currently held by T_2, so on and so forth. Note that this can lead to deadlocks as well.

To summarize, the main issues that arise out of priority inversion related phenomena are unbounded priority inversion, second blocking and chain blocking. Coupled with known issues like deadlocks, we need to design protocols such that all three scenarios are prevented by design.

Definition 5.5.3 Unbounded Priority Inversion and Chain Blocking

When a high-priority task is blocked for an unbounded amount of time because the low-priority task that holds the resource is continuously preempted, this phenomenon is known as unbounded priority inversion. Next, assume a task that needs to acquire many resources. If after acquiring the first resource, it waits to acquire any subsequent resource, then this phenomenon is known as second blocking.

If task T_1 is blocked by T_2, which in turn is blocked by T_3, then we have chain blocking.

Let us make a set of crucial assumptions here (refer to Point 5.5.2).

Point 5.5.2

- Let the pri_c function denote the current priority of a task. For example, $pri_c(T)$ is the instantaneous priority of task T. Let its original priority be denoted by the expression $pri_o(T)$. Note that $pri_c(T)$ can change over time. Let us assume that the initial pri-

orities assigned to tasks are fully comparable – no two priorities are equal. This can easily be accomplished by breaking ties (same priority numbers) using task numbers.

- Assume a uniprocessor system.

The Priority Inheritance Protocol (PIP)

The simplest protocol in this space is known as the priority inheritance protocol (PIP), which is supported in current Linux versions. The key idea is to raise the priority of the resource-holding task temporarily, even though it may be a low-priority task. It is set to a value that is equal to the priority of the high-priority resource-requesting task. This is done to ensure that the resource-holding task can finish quickly and release the resource, regardless of its original priority. It is said to *inherit* the priority of the resource-requesting task.

Let us explain priority inheritance in some more detail. There are two cases here. Let the priority of the resource-holding task T_{hld} be p_{hld} and the priority of the resource-requesting task T_{req} be p_{req}. If $p_{hld} < p_{req}$, we temporarily raise the priority of T_{hld} to p_{req}. However, if $p_{hld} > p_{req}$, nothing needs to be done. Note that this is a *temporary action*. Once the contended resource is released, the priority of T_{hld} reverts to p_{hld}. Now, it is possible that p_{hld} may not be the original priority of T_{hld} because this itself may be a *boosted priority* that T_{hld} may have inherited because it held some other resource. We will not be concerned about that and just revert the priority to the value that existed just before the resource was acquired, which is p_{hld} in this case.

Note that a task can inherit priorities from different tasks in the interval of time in which it holds a resource. Every time a task is blocked because it cannot access a resource, it tries to make the resource-holding task inherit its priority if there is a case of inversion.

Let us explain with an example. Assume that the real-time priority of the low-priority task T_{low} is 5. The priority of a medium-priority task T_{med} is 10, and the priority of the high-priority task T_{high} is 15. These are all real-time priorities: higher the number, greater the priority. Now assume that T_{low} is the first to acquire the resource. Next, T_{med} tries to acquire the resource. Due to priority inheritance, the priority of T_{low} now becomes 10. Next, T_{high} tries to acquire the resource. The priority of T_{low} ends up getting boosted again. It is now set to 15. After releasing the resource, the priority of T_{low} reverts back to 5.

This is a very effective idea, and it is simple to implement. This is why many versions of Linux support the PIP protocol. Sadly, the PIP protocol suffers from deadlocks and chain blocking. It only offers a solution for unbounded priority inversion. Let us explain.

In this context, we shall specifically define two types of blocking: resource blocking and inheritance blocking. Resource blocking happens when one task is trying to acquire a resource that is held by another task. In some cases, a resource acquisition request may be denied because the priority of the requesting task is not high enough. All such cases qualify as resource blocking. Inheritance blocking means that the current task cannot execute on the CPU because other

tasks have boosted their priority.

Issues with the PIP Protocol

Let the low-priority task be T_{low} and the high-priority task be T_{high}. Assume T_{low} has acquired resource R_1, it is preempted by T_{high}, T_{high} acquires R_2 and then tries to acquire R_1. It then blocks. T_{low} resumes and tries to acquire R_2. We have a deadlock.

In this case, unbounded priority inversion is not possible. This is because the low-priority task that holds the resource does not remain "low priority" anymore once a high-priority task waits for it to complete. In this case, the priority of T_{low} is boosted. It is thus not possible for any medium-priority task to preempt T_{low} and indefinitely block T_{high} in the process.

Sadly, second blocking is possible. Once a resource has been acquired, acquiring any subsequent resource requires the task to wait for another task to release it. Assume two resources R_1 and R_2. R_1 is held by T_1 and R_2 is held by T_2. A third task T_3 can try to acquire R_1 and then R_2 in order. It will get blocked both the times.

> **Point 5.5.3**
>
> Deadlocks, chain blocking and second blocking are the major issues in the PIP protocol.

5.5.6 Highest Locker Protocol (HLP)

Let us now try to implement a better protocol called the Highest Locker Protocol (HLP) that solves some of the aforementioned problems. Assume that some facts are known a priori such as the resources that a task may acquire. Moreover, let us define a $ceil(resource)$ function, which is defined as the priority of the highest priority task that can *possibly acquire* a resource (some time in the future).

The Algorithm

Once a task acquires a resource, we raise its priority to $ceil(resource) + 1$, if this value is higher than the task's existing priority. This may be perceived to be unfair. In this case, we are raising the priority to an absolute maximum, which is more than the original priority of any high-priority task that covets the resource. Essentially, this is priority inheritance on steroids !!!

Akin to the PIP protocol, unbounded priority inversion is not possible because no medium-priority task can block the resource-holding task. In fact, any priority inheritance or priority boosting protocol will avoid unbounded priority inversion because the priority of the resource-holder becomes at least as large as the resource-requester. The resource holder thus cannot be preempted by any intermediate-priority process.

Resource Blocking

> **Theorem 5.5.1** No resource blocking in the HLP protocol
>
> There is no resource blocking in the HLP protocol.

Proof: Assume that there is resource blocking. Consider the first such instance in the system. Let T_1 get blocked while trying to acquire R at t_{block}. This is because R has been acquired by T_2 already.

The question is why is T_2 not running at t_{block}? Consider the time instant at which T_2 just finished acquiring R. At that point of time (t_0), either T_1 had not started or the following relationship was true: $pri_c(T_2) > pri_c(T_1)$. If T_1 had not started, then it would have not gotten a chance to execute. This is because $pri_o(T_1) \leq ceil(R) < pri_c(T_2)$. This means that T_2 will not allow T_1 to start executing unless it gets blocked, which will not happen before t_{block}.

Consider the case: $pri_c(T_2) > pri_c(T_1)$ at t_0. T_2 could not have been blocked before t_{block}. The question in this case is how did T_1 get a chance to run when T_2 was alive? This is possible if a high-priority task preempted T_2. That is the only way in which T_1 could have acquired a high priority via the priority inheritance mechanism. This would require the high-priority task to get blocked to allow T_1 to acquire its priority. This is not possible because no task can get blocked before t_{block}. Hence, it is not possible for T_1 to have run before T_2 finished. We thus have a contradiction here. ∎

> **Lemma 1**
>
> There are no deadlocks in the HLP protocol.

Proof: Deadlocks happen because a task holds on to one resource, and tries to acquire another resource unsuccessfully (hold-and-wait condition). Given that there is no resource blocking in HLP (Lemma 5.5.1), this condition will never be realized. Given that the hold-and-wait condition is one of the necessary conditions for a deadlock, a deadlock will never form. ∎

> **Lemma 2**
>
> Chain blocking is not possible in the HLP protocol.

Proof: Consider a dependence chain $T_1 \rightarrow T_2 \rightarrow T_3$. Consider task T_2 that has already started running and acquired a resource. It cannot get resource-blocked (see Lemma 5.5.1). Hence, the $T_2 \rightarrow T_3$ dependency will not be there. Hence, chain blocking is not possible in the HLP protocol. ∎

> **Lemma 3**
>
> Freedom from chain blocking implies deadlock freedom.

Proof: A deadlock implies that there is a circular wait of the form $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_1$. This means that there is a chain blocking pattern in the system. We can thus say that a deadlock implies chain blocking. Alternatively, we can state the contrapositive, "freedom from chain blocking implies deadlock freedom". ∎

Inheritance Blocking

This protocol however does create an additional issue of inheritance blocking. Assume that the priority of task T is 5 and the resource ceiling is 25. In this case, once T acquires the resource, its priority becomes 26. This is very high because 25 is a hypothetical maximum that may get realized very rarely. Because of this action, all the high-priority tasks with priorities between 6 and 25 get preempted. The sad part is that there may be no other process that is interested in acquiring the resource regardless of its priority. We still end up stopping a lot of processes from executing.

> **Point 5.5.4**
>
> Inheritance blocking is the major issue in the HLP protocol. HLP does not suffer from resource blocking, chain blocking, second blocking, deadlocks and unbounded priority inversion.

5.5.7 Priority Ceiling Protocol (PCP)

Let us next look at the Priority Ceiling Protocol (PCP), which tries to rectify some issues with the HLP protocol. Here also every resource has a resource ceiling. However, the key twist here is that we may not allocate a resource to a task, even if it is *free*. This may sound inefficient, but it will help us guarantee many important properties.

In PCP, the current system ceiling or CSC is defined as the maximum of all the ceilings of all the resources that are currently *acquired* by some task in the system. PCP has a "resource grant clause" and "inheritance clause". The latter changes the priority of the task.

Resource Grant and Inheritance Clauses

Let us outline the two basic rules that determine the behavior of the PCP protocol. The resource grant clause specifies the set of rules that the OS or resource manager follows to grant a resource to a task.

Resource Grant Clause For task T to acquire a resource R any of the two conditions must be true.

1. The original priority of T is greater than the CSC.

2. T holds a resource that set the current system ceiling.

Inheritance Clause The task holding a resource *inherits* the priority of the blocked task, if its priority is lower. This is the same as the priority inheritance protocol.

Let us understand the resource grant clause in some further detail. Let us call a resource that has set the CSC a *critical resource*. This means that when it was acquired the value of the system-wide CSC increased. It was set to a new value. If a task T owns a critical resource, then the resource grant clause allows it to acquire additional resources at will. There are two cases that arise after the resource has been acquired. Either the existing critical resource continues to remain critical or the new resource that is going to be acquired becomes critical. In both cases, T continues to <u>own</u> the critical resource.

In the other sub-clause, a task can acquire a resource if its original priority is greater than the CSC. This clause is beneficial when the task has not acquired any resources yet. Otherwise, its priority would not have been greater than the CSC.

Lemma 4

The moment a task whose original priority is greater than the CSC acquires a resource, it sets the CSC and that resource becomes *critical*.

The inheritance clause is similar to the inheritance mechanism of the PIP protocol.

Properties of the PCP Protocol

Given that this protocol has priority inheritance (similar to PIP), unbounded priority inversion is not possible. Let us next look at deadlocks and chain blocking.

Let us first prove that second blocking is not possible. This means that it is not possible for a task to acquire a resource and wait to acquire another resource.

Lemma 5

Assume task T acquires resource R at time t because its original priority is higher than the CSC. Let the set $\mathcal{S}$ denote the set of all the tasks that have already acquired resources before t. If it is guaranteed that T will not block, then it is not possible for any future task to be interested in a resource acquired by a task in S.

Proof: T will continue to execute after acquiring R until it is preempted by a higher priority process T_H. Note that T_H has to be a fresh process. It cannot be a process that was in the system before. This is because no event happened to increase its priority.

Now T_H cannot be interested in any resource acquired by a task in $\mathcal{S}$. This is because $CSC(t^-) \geq pri_o(T_H)$. We know that $pri_c(T) > CSC(t^-)$. Hence, $pri_c(T) > pri_o(T_H)$, which will not allow T_H to preempt T. There is a contradiction here.

We can extend the argument to prove that if T_H is preempted by another higher priority process, then it too cannot covet any resource acquired by tasks in $\mathcal{S}$. ■

> **Theorem 5.5.2** PCP does not suffer from second blocking
>
> If a task acquires one resource, it will not get blocked while acquiring any other resource. Alternatively, second blocking is not possible in the PCP protocol.

Proof: Assume that the protocol runs for some time without second blocking. Then at time t_{block} the first instance of second blocking is recorded. This happens when task T_1 that started out with acquiring R_1, gets blocked while trying to acquire R_2. This is because R_2 has already been acquired by T_2. Let the time at which T_1 acquired R_1 be t_1.

Case: I T_2 acquired R_2 before t_1: In this case, the system ceiling is set to at least $ceil(R_2)$. This means that $CSC(t_1) \geq ceil(R_2)$. Given that T_1 is interested in R_2, $pri_o(T_1) \leq ceil(R_2) \leq CSC(t_1)$. T_1's original priority is not more than CSC. It could not have acquired R_1 at t_1. Note that at this point of time it did not have a resource that had set the system ceiling.

Case: II T_2 acquired R_2 after t_1: At t_1, T_1 acquires R_1. $\therefore CSC(t_1) \geq ceil(R_1)$. Given that T_2 acquired R_2, $pri_o(T_2) > CSC(t_1) \geq ceil(R_1) \geq pri_o(T_1)$. Note that T_2 cannot get blocked until t_{block}. The only way that T_1 can start executing is if its priority is somehow increased. This can only happen via priority inheritance. This means that some ultra-high-priority task T_H needs to block on some resource acquired by T_1. This will happen only if T_H arrives after T_2 has acquired R_2. Using the results in Lemma 5, we can prove that no such task T_H can covet any resource acquired in the system by existing tasks. Hence, it is clear that T_1 will not get a chance to execute and thus it will not be able to block on R_1.

We thus arrive at a contradiction. Hence, we prove that second blocking is not possible in the PCP protocol. ■

> **Lemma 6**
>
> If there is no second blocking, it means that there is no chain blocking.

Proof: In chain blocking, we have a situation like this: $T_1 \rightarrow T_2 \rightarrow T_3$. Clearly, in this case, T_2 has acquired the resource that T_1 wants to lock first, and then it is waiting on T_3 to release its resource. This is a case of second blocking. This means that chain blocking implies second blocking.

The contrapositive is that no second blocking implies no chain blocking. ■

This means that the PCP protocol does not have chain blocking. Using the results of Lemma 3, we can say that PCP does not suffer from deadlocks.

Next, note that in the PCP protocol we do not elevate the priority to very high levels, as we did in the HLP protocol. The priority inheritance mechanism is the same as the PIP protocol. Hence, we can conclude that inheritance blocking is far more controlled.

Point 5.5.5

The PCP protocol does not suffer from deadlocks, second blocking, chain blocking and unbounded priority inversion. The problem of inheritance blocking is also significantly controlled.

5.6 Summary and Further Reading

5.6.1 Summary

Summary 5.6.1

1. A data race is defined as a pair of conflicting and concurrent accesses to a single global shared variable.

    (a) Two accesses are said to be conflicting if one of them is a write.

    (b) Two accesses are concurrent if there is no path between them that has a synchronization operation.

2. To eliminate data races and consequent non-intuitive behavior, it is important to properly label programs. This is done by encapsulating shared variable accesses with lock/unlock operations such that every shared variable is protected by the same lock.

3. Locks enforce mutual exclusion – it is not possible for two threads to execute a critical section at the same point of time.

4. They introduce an additional problem of deadlocks that form when four conditions are met:

    Hold and wait A process holds a few locks and is waiting to acquire a few more locks.

    No preemption This basically means that locks and their corresponding resources cannot be forcefully taken away from processes.

    Mutual exclusion A lock cannot be concurrently acquired by two processes.

    Circular wait The processes wait on each other. There is a cyclic wait: $A \rightarrow B \rightarrow C \rightarrow \ldots A$.

5. Atomic operations can be used to perform read-modify-write operations such as atomic increment. They appear to complete instantaneously.

6. Atomic operations are expensive in terms of performance. Sadly, using regular reads and writes instead of them is tricky. Compilers and processors tend to reorder memory operations. A memory

consistency model specifies the *valid* outcomes of a parallel program based on the allowed reorderings.

7. A sequentially consistent (SC) memory consistency model implies atomicity and per-thread ordering. Atomicity means that every operation appears to complete instantaneously. Per-thread ordering means that all the memory operations issued by the same thread appear to complete in the order in which they are issued.

8. All memory consistency models have fence instructions that enforce an ordering. Most atomic instructions are are also synchronizing instructions (with in-built) fences. Lock and unlock primitives use them.

9. If a program is properly labeled and lock/unlock operations have synchronization instructions, then it can be proven that a data-race-free program always produces sequentially consistent executions.

10. There are three kinds of non-blocking algorithms that use atomic operations instead of locks.

    Wait-free Every operation completes within a bounded number of internal steps.

    Lock-free If all the threads in the system complete a certain number of cumulative internal steps, at least one of the threads is guaranteed to complete its operation.

    Obstruction-free If the rest of the threads go to sleep, then the sole running thread is guaranteed to complete its operation within a bounded number of internal steps.

11. Semaphores are a generalized version of locks. To acquire a semaphore, the internal count variable needs to be decremented. If the count is zero, then the thread waits for it to be non-zero.

12. Reader-writer locks allow either multiple readers or a single writer.

13. Queues are integral to most kernel data structures. If there is a single producer and single consumer, then it is easy to write a wait-free queue. Generic versions that allow multiple producers and consumers require locks and semaphores to enforce the correctness of concurrent executions. Semaphores are naturally suitable for implementing bounded queues.

14. The kernel defines a spinlock that acts as a lock on the CPU. It requires busy waiting and can be used in an interrupt context.

15. Spinlocks are used to create kernel mutexes that can be used in all kernel contexts. They have a fast path where the mutex can be directly acquired using a compare-and-swap (CAS) primitive. If it cannot be acquired using the fast path, then there is a slow path in which the process is added to a queue of waiting processes.

16. The RCU (read-copy-update) mechanism is integral to the Linux kernel. It allows readers to concurrently access data without acquiring locks. Writes always make updates to a new copy of data and ultimately replace the old copy with the new copy. Once all the readers who were accessing the old copy have completed their execution, the old copy is garbage collected.

17. The KSW model defines a space of scheduling problems.

    (a) Shortest jobs first (SJF) is the most optimal algorithm for $1 \mid\mid C_j$.

    (b) A weighted version of the problem has an optimal solution if the priority is set to w_j/p_j (weight divided by the execution time).

    (c) The EDF algorithm produces the most optimal schedule for the problem $1 \mid r_i, dl_i, pmtn \mid L_{max}$. It minimizes the lateness when preemption is enabled and jobs can arrive anytime.

    (d) The SRTF (shortest remaining time first) produces the most optimal schedule for $1 \mid r_i, pmtn \mid \sum C_i$.

    (e) A lot of variants of the scheduling problem are NP-complete.

18. In the space of multicore scheduling, list scheduling is a popular algorithm that schedules tasks across the cores in descending order of priority subject to the fact that there is no deliberate idling. It is possible to prove that the ratio between C_{max} produced by list scheduling and the optimal value of C_{max} is bounded by $2 - 1/m$, in a system with m CPUs. The bound can be improved to $\frac{4}{3} - \frac{1}{3m}$, if we arrange jobs in descending order of execution times.

19. The Banker's algorithm is used to avoid and detect deadlocks in systems with multiple copies of resources.

20. The Linux kernel has different scheduling classes. It always runs tasks in a higher scheduling class before scanning queues of a lower scheduling class.

    Stop This is the highest priority class. For example, if there is a kernel panic or a new CPU is added, tasks in this class are created.

    Deadline Uses the EDF scheduler to implement tasks that have explicit deadlines.

    Real Time Can use FIFO or round-robin scheduling to schedule real-time tasks.

    CFS The Completely Fair Scheduler (CFS) is used to schedule regular non-real-time tasks. It uses the notion of the virtual runtime (vruntime) that is a function of the actual runtime and the task's priority. The aim is to ensure that every task gets at least one chance to execute in a scheduling period, and

all the tasks accumulate the same vruntime in a scheduling period.

Idle This is a task that runs when there are no active tasks that are runnable. It is mainly used for accounting and bookkeeping purposes. In most cases, it simply puts the CPU to sleep until it is woken up by an interrupt.

21. We typically consider periodic tasks in real-time scheduling.

    (a) EDF produces feasible schedules as long as the utilization is less than or equal to 1. It uses dynamic priorities that keep getting recomputed based on the arrival of new tasks and their associated deadlines.

    (b) RMS (Rate Monotonic Scheduling) uses static priorities that do not change throughout the execution.

        i. The Liu-Layland bound is a sufficient condition for schedulability. It says that the system is schedulable if the utilization $U \leq n(2^{\frac{1}{n}} - 1)$, where there are n tasks in the system. If $n \to \infty$, U tends to 0.69 (ln(2)).

        ii. The Lehoczky's test produces tighter bounds and includes some necessary conditions as well.

    (c) In both RMS and EDF, the deadline is the same as the period. However, in the general case, the deadline could be smaller than the period. In this case, Deadline Monotonic Scheduling (DMS) is used where the priority is inversely proportional to the deadline (relative to the job arrival time).

22. The priority inheritance protocol ensures that there is no unbounded priority inversion. This is ensuring by temporarily assigning the priority of the higher priority blocked thread to the lower priority thread that has blocked it. When the resource is released, the priority reverts back to the process's priority that existed before the resource was acquired. This protocol can lead to deadlocks and chain blocking (repeated blocking).

23. The Highest Locker Protocol (HLP) uses the notion of a ceiling of a resource, which is the highest priority of any task that may access the resource at some point in the future. Whenever, a resource R is acquired, the priority of the process is set to $max(pri, ceil(R)+1)$, where pri is the current priority and $ceil$ is the resource ceiling function. This ensures that the lock-holding process executes at a very high priority. This protocol successfully avoids chain blocking and deadlocks. However, it introduces a new problem of inheritance blocking, where the priority becomes so high that intermediate priority processes do not get a chance.

24. The Priority Ceiling Protocol (PCP) solves all the aforementioned problems including inheritance blocking by defining a system ceiling (CSC), which is the highest resource ceiling of any resource

> that has currently been acquired by any process in the system. A process can only acquire a resource if it has either set the current CSC or its priority is greater than the current CSC. This means that even if a resource is free, a process may not be allowed to acquire it. This is an altruistic choice; however, it produces good outcomes at the level of the entire system. No priority is boosted to a very high value (as in the HLP protocol) and thus intermediate priority processes get a fair chance to execute.

5.6.2 Further Reading

Exercises

Ex. 1 — What are the four necessary conditions for a deadlock? Briefly explain each condition.

Ex. 2 — Assume a system with many short jobs with deterministic execution times. Which scheduler should be used?

Ex. 3 — Design a concurrent stack using the compare-and-set(CAS) primitive. Use a linked list as a baseline data structure to store the stack (do not use array).

```
int CAS (int *location, int old_value, int value) {
    if (*location == old_value) {
        *location = value;
        return 1;
    } else return 0;
}
```

Do not use any locks (in any form). In your algorithm, there can be starvation; however, no deadlocks. Provide the code for the `push` and `pop` methods. They need to execute atomically. Note that in any real system there can be arbitrary delays between consecutive instructions.

Ex. 4 — Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.

Ex. 5 — Propose a solution to the Dining Philosopher's problem that is starvation-free?

Ex. 6 — Solve the Dining Philosopher's Problem using only semaphores. Use three states for each philosopher: THINKING, HUNGRY and EATING.

Ex. 7 — Acquiring a mutex is a complex process. We have a fast path and different variants of slow paths. Why do we need so many paths? Explain with examples.

Ex. 8 — Consider the kernel mutex. It has an owner field and a waiting queue. A process is added to the waiting queue only if the owner field is populated (mutex is busy). Otherwise, it can become the owner and grab the mutex. However, it is possible that the process saw that the owner field is populated, added itself to the waiting queue but by that time the owner field became empty – the previous mutex owner left without informing the current process. There is thus no process to wake it up now, and it may wait forever. Assume that there is no dedicated thread to wake processes up. The current owner wakes up one waiting process when it releases the mutex (if there is one).
Sadly, because of such race conditions, processes may wait forever. Design a kernel-based mutex that does not have this problem. Consider all race conditions. Assume that there can be indefinite delays between instructions. Try to use atomic instructions and avoid large global locks. Assume that task ids require 40 bits.

Ex. 9 — The Linux kernel has a policy that a process cannot hold a spinlock while attempting to acquire a semaphore. Explain why this policy is in place.

Ex. 10 — Why are semaphores stronger synchronization primitives than condition variables and similar user-space synchronization mechanisms?

Ex. 11 — Explain the spin lock mechanism in the Linux kernel (based on ticket locks). In the case of a multithreaded program, how does the spin lock mechanism create an order for acquiring the lock? Do we avoid starvation?

Ex. 12 — What will it take to implement linearizability at the hardware level? How will OS code get easier if the hardware provides linearizability? Explain with examples and justify your answer.

Ex. 13 — Why are memory barriers present in the code of the `lock` and `unlock` functions?

Ex. 14 — Write a fair version of the reader-writer lock.

Ex. 15 — What is the lost wakeup problem? Explain from a theoretical perspective with examples.

Ex. 16 — Does the Banker's algorithm prevent starvation? Justify your answer.

Ex. 17 — We wish to reduce the amount of jitter (non-determinism in execution). Jitter arises due to interrupts, variable execution times of system calls and the kernel opportunistically scheduling its own work when it is invoked. This makes the same program take different amounts of time when it is run with the same inputs. How can we create an OS that reduces the amount of jitter? What are the trade-offs?

* **Ex. 18** — Implementing lottery scheduling as follows. Assign lottery tickets (unsigned integers) to processes. Whenever a scheduling decision needs to be

made, a lottery ticket is chosen at random, and the process holding that ticket gets the CPU. Describe how such a scheduler can ensure that higher-priority threads receive more attention from the CPU than lower-priority threads, yet not severely compromise on fairness.

Ex. 19 — Is it guaranteed that we will resume the process that invoked a system call after the system call is serviced? Why or why not? What factors should we consider?

Ex. 20 — Prove that no reader can be alive when `synchronize_rcu` returns. Create diagrams with happens-before edges, and prove that such a situation is not possible. Show a proof by contradiction.

Ex. 21 — Show the pseudocode for registering and deregistering readers, and the `synchronize_rcu` function.

Ex. 22 — How is preemption enabled and disabled?

Ex. 23 — Why is it advisable to use RCU macros like `rcu_assign_pointer` and `rcu_dereference_check`? Why cannot we read or write to the memory locations directly using simple assignment statements?

Ex. 24 — Consider the following code snippet.

```
struct foo {
     int a;
     int b;
     int c;
};
struct foo *gp = NULL;

/* . . . */
struct foo *p = kmalloc(sizeof(struct foo), GFP_KERNEL);
        /* kernel malloc */
p->a = 1;
p->b = 2;
p->c = 3;
```

Write the code to set `gp` to `p`.

Ex. 25 — Correct the following piece of code in the context of the RCU mechanism.

```
p = gp;
if (p != NULL) {
    myfunc (p->a, p->b, p->c);
}
```

Ex. 26 — Users working on a laptop or desktop typically interact with a few tasks via graphical interfaces. These could be games, web browsers or media players. Sometimes it is necessary to boost their priority for a better user experience. Suggest how this can be done.

Ex. 27 — How can we modify the CFS scheduling policy to fairly allocate processing time among all users instead of processes? Assume that we have a single CPU and all the users have the same priority (they have an equal right to the CPU regardless of the processes that they spawn). Each user may spawn multiple processes, where each process will have its individual CFS priority between 100 and 139. Do not consider the real-time or deadline scheduling policies.

Ex. 28 — How does the Linux kernel respond if the current task has exceeded its allotted time slice?

Ex. 29 — The process priorities vary exponentially with the nice values. Why is this the case? Explain in the context of a mix of compute and I/O-bound jobs where the nice values change over time.

* **Ex. 30** — Given a mixture of interactive, I/O-intensive and long-running processes whose execution time is not known a priori, design a scheduling algorithm for a single CPU that optimizes the completion time as well as the responsiveness of interactive jobs. The algorithm (and associated data structures) should take into account the diversity of jobs and the fact that new jobs and high-priority jobs need quick service, whereas low-priority long-running batch jobs can be delayed (read deprioritized).

* **Ex. 31** — Prove that $(1 \mid r_i \mid L_{max})$ is NP-complete.

** **Ex. 32** — Prove that the competitive ratio is bounded by $(\frac{4}{3} - \frac{1}{3m})$ if we schedule processes in descending order of processing times.

** **Ex. 33** — Prove that any algorithm that uses list scheduling will have a competitive ratio (C_{list}/C^*), which is less than or equal to $(2 - 1/m)$. There are m processors, C is the makespan and C^* is the optimal makespan.

Ex. 34 — For a system with periodic and preemptive jobs, what is the utilization bound (maximum value of U till which the system remains schedulable) for EDF?

Ex. 35 — Prove that in PCP algorithm, once the first resource is acquired, there can be no more priority inversions (provide a very short proof).

Chapter 6

The Memory System

The operating system is a resource manager at its core. In the previous chapter, we focused on scheduling and synchronization. The CPU was the resource, which was being managed. We shall focus on managing physical and virtual memory in this chapter. Both of them are legitimate resources. The virtual memory is a complex entity, which is split between a user process and the kernel. We shall study in this chapter that the kernel virtual memory space is further split into many different regions. The physical memory is also a large and complex entity that comprises the memory modules, the swap space and parts that interact with DMA engines and I/O devices.

Recall the discussion in Chapter 2. We had argued that the most important functionality provided by the virtual memory subsystem in the kernel is isolating different processes. The virtual memory mechanism ensures that no process can access the memory space of any other process without proper authorization. We shall see in this chapter that apart from correctness concerns, there are other issues with regards to efficiency and proper TLB management. Up till now we have been considering the TLB to just be a simple cache that stores frequently used mappings. The only TLB manipulation instructions that we were aware of were adding entries to the TLB and flushing all its entries, especially when the page table is reloaded. In this chapter, we shall look at a more nuanced picture. It is possible to pin entries, annotate entries with the process id and selectively flush entries. All of these facilities are needed for performance efficiency.

When it comes to managing physical memory, page replacement algorithms are arguably the most important determinants of performance. Hence, we need to ensure that the page replacement algorithm is as good as possible. The aim is to minimize page faults. Finding candidate pages for replacement requires maintaining a lot of data structures and doing a lot of bookkeeping. There is clearly a need to move a lot of this work off the critical path. There is a lot of theoretical work in this space. Different algorithms have been proven to be optimal under different types of constraints. Such algorithms are typically quite simple. They are sadly not potent enough to be used in a large and complex system such as the Linux kernel.

There is a need to design a bespoke algorithm that has low overheads, is efficient and handles all kinds of special cases. In the current version of the

Linux kernel, we typically manage single pages, folios that contain N pages where N is a power of 2, and huge pages (a single page is either 2 MB or 1 GB). Furthermore, pages can be regular memory pages or could be memory-mapped pages that are backed up by I/O devices. Some pages can be written to by the CPUs and also enable DMA access. Therefore, any page management and replacement algorithm needs to keep all these things in mind. In this context, we will introduce the multi-generation MGLRU algorithm. Its basic philosophy is quite simple yet it is powerful enough to handle realistic scenarios.

In the context of page replacement, the issue of *reverse mapping* also becomes important. For a given physical page, it is necessary to maintain a list of all the virtual pages that map to it. For each virtual page, we need to maintain a pointer to its page table and `task_struct`. If the physical page is evicted from main memory, then the mappings corresponding to all the virtual pages need to be changed. We need to record the fact that the mapped physical page is no more in physical memory. Things get further complicated if a process is forked. This means that a physical page has more sharers. Furthermore, it is possible that the parent and child processes can get forked several times subsequently. There is thus a need to manage a large number of sharers per physical page. Their copy-on-write status needs to be tracked.

Managing kernel memory is equally challenging. It is quite different from allocating memory in the user space. There is a need to create bespoke mechanisms in the kernel for managing its memory. We cannot allow kernel processes to allocate arbitrary amounts of memory or have very large data structures whose sizes are not known or not bounded – this will create a lot of problems in kernel memory management. There is a need to manage kernel memory more deterministically. It is thus not a good idea to allocate objects in the kernel the same way that we do in user space where regions do not have strict bounds in terms of size. There are also no requirements for the regions to be contiguous. It is wiser to follow an intelligent version of a simple base-limit scheme. In this context, we shall discuss several memory allocation and management schemes.

Organization of this Chapter
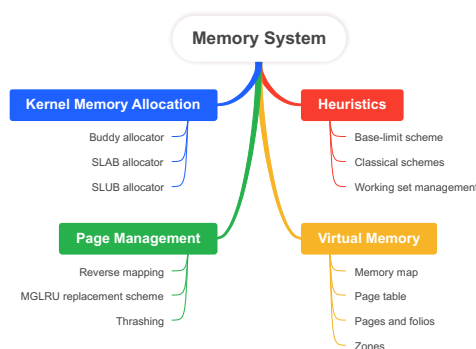
Figure 6.1: Organization of this chapter

Figure 6.1 shows the organization of this chapter.

We shall start with a section on memory management heuristics. Here, we will discuss classical memory management schemes such as the base-limit scheme. They are not used in modern kernels. However, a lot of contemporary schemes draw inspiration from them. Hence, appreciating them is a worthwhile exercise.

Next, we shall take an in-depth look at virtual memory. The kernel's memory map is large and quite elaborate. There are a lot of dedicated regions for storing diverse types of information. The page table is also quite complex. It uses intricate bitwise operations to speed up page walking.

Note that paging the entire memory breaks the notion of contiguity of physical memory addresses. Sometimes, there is a need to create contiguous regions in the physical memory space for effective management, prefetching and effective metadata management. On similar lines, it is a wise idea to partition physical memory into zones. Each zone is specialized for different kind of memory accesses.

Subsequently, we shall look at page management. We shall look at the issues related to replacement and reverse mapping together. Reverse mapping will be discussed first and then the MGLRU algorithm will be introduced. A problem that can arise is thrashing, which can crash the entire system.

Finally, we shall look at memory allocation in the kernel. We shall discuss all the three popular memory allocators: buddy allocator, the Slab and Slub allocators.

6.1 Traditional Heuristics for Page Allocation

6.1.1 Base-Limit Scheme

Let us consider traditional heuristics for memory management. We need to go back to the era when virtual memory did not exist. Consider systems that do not have virtual memory such as small embedded devices. Clearly there is a need to isolate process address spaces. This is achieved with the help of two registers namely *base* and *limit*. As we can see in Figure 6.2, the memory for a process is allocated contiguously. The starting address is stored in the *base* register and after that the size of the memory that the process can access is stored in the *limit* register. Any address issued by the processor is translated to the physical address by adding it to the *base* register. If the issued address is A, then the address sent to the memory system is $base + A$. It is checked to see if it is less than *limit* or not. If it is less than *limit*, then the address is deemed to be correct. Otherwise, the memory address is declared to be out of bounds.

We can visualize the memory space as a sequence of contiguously allocated regions (see Figure 6.2). There are *holes* between allocated regions. If a new process is created, then its memory requirement needs to be known a priori. The memory needs to be allocated within one of the holes. Let us say that a process requires 100 KB and the size of a hole is 150 KB, then we are leaving 50 KB free. By following this process, we basically create a new hole that is 50 KB long. This phenomenon of having holes between regions and not using that space is known as *external fragmentation*. On the other hand, leaving space empty within a page in a regular virtual memory system is known as *internal*

fragmentation.

Definition 6.1.1 Fragmentation

Internal Fragmentation It refers to the phenomenon of leaving memory space empty within pages in a virtual memory system. The wastage per page is limited to 4 KB per page.

External Fragmentation It is relevant in the context of a base-limit addressing system where space in the memory system is kept empty in the form of holes.
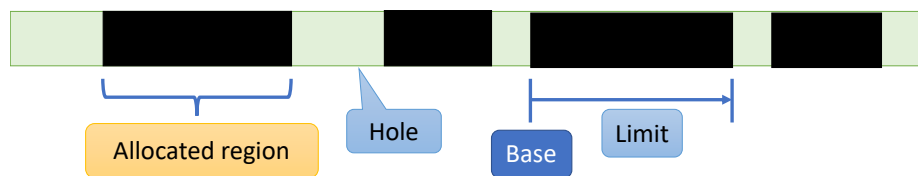
Figure 6.2: Memory allocation with *base* and *limit* registers

Let us proceed with the assumption that we are precisely aware of the maximum amount of memory that a new process requires. We need to select an appropriate hole for allocating memory. Clearly the size of the hole needs to be more than the amount of requested memory. There could be multiple such holes, and we need to choose one of them. Our choice really matters because it determines the efficiency of the entire process. It is indeed possible that later on we may not be able to satisfy requests primarily because we will not have holes of adequate size left. Hence, designing a proper heuristic in this space is important in anticipation of the future. There are several heuristics in this space.

Assume that we need R bytes for a new process.

Best Fit Choose the smallest hole that is just about larger than R.

Worst Fit Choose the largest hole.

Next Fit Start searching from the last allocation that was made and move towards higher addresses (with wraparounds).

First Fit Choose the first available hole

These heuristics perform very differently for different kinds of workloads. For some workloads, they perform really well whereas for many other workloads their performance is quite below par. It is also possible to prove that they are optimal in some cases assuming some simple distribution of memory request sizes in the future.

The fact still remains that in general we do not know how much memory a process requires. Hence, assessing or declaring the amount of memory that a process requires upfront is quite difficult. Any such estimate is bound to be quite conservative. Note that this information is not there with the compiler

or even the user. In today's complex programs, the amount of memory that is going to be used is a very complicated function of the input, and it is thus not possible to predict it beforehand. As a result, these schemes are seldom used as of today. They are nevertheless relevant for very small embedded devices that cannot afford virtual memory. However, by and large, the base-limit scheme is consigned to the museum of virtual memory schemes.

6.1.2 Classical Schemes to Manage Virtual Memory

Managing memory in a system that uses virtual memory is relatively straightforward. To keep track of free frames (physical pages) in memory, we can use a bit vector – one bit for each frame. Next, to find the first free frame, we can accelerate the process using an augmented tree (see Appendix C). In $log(N)$ time, we can find the first free frame and allocate it to a process. Even freeing frames is quite easy in such a system that uses a bit vector and an augmented tree.

The most important problem in this space is finding the page that needs to be replaced in case the memory is full. This has a very important effect on the overall performance because it affects the page fault rate. Page faults necessitate expensive reads to the underlying storage device: hard disk or flash drive. This process requires millions of cycles. Hence, page faults are regarded as one of the biggest performance killers in modern systems.

The Stack Distance

To understand the philosophy behind replacement algorithms, let us understand the notion of stack distance (refer to Figure 6.3). We conceptually organize all the physical pages (frames) that have been accessed till a given point of time in a stack. Whenever a page is accessed, it is placed at the top of the stack. If it was already present in the stack at a different location, it is removed from there. The distance between the top of the stack and the point in the stack where the page was found is known as the *stack distance*.

Assume that a page is accessed twice. The second time the stack distance is 0 because the page is now at the stack top.

> **Definition 6.1.2** Stack Distance
>
> We maintain a stack of all the pages that have ever been accessed. Whenever a page is accessed, we record the distance between the stack top and the point at which the page was found in the stack. This is known as the stack distance. Then we move the page to the top of the stack. If a page is being accessed for the first time, then it is simply placed at the top of the stack. The stack distance is not recorded.
> The stack distance distribution is a standard tool used to evaluate temporal locality in computer systems ranging from caches to paging systems.

The stack distance typically has a distribution that is similar to the one shown in Figure 6.4. Note that we have deliberately not shown the units of the x and y axes because the aim is to just show the shape of the curve and not focus on specific values. We observe a classic heavy-tailed distribution where small values are relatively infrequent. Then there is a peak followed by a very
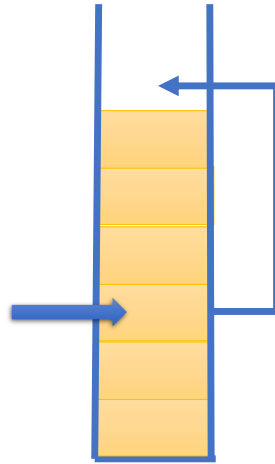
Figure 6.3: Notion of the stack distance

heavy tail. The *tail* basically refers to the fact that we have non-trivially large probabilities when we consider rather high values of the stack distance.

This curve can be interpreted as follows. Low values of the stack distance are relatively rare. This is because we typically tend to access multiple streams of data simultaneously. We are definitely accessing data as well as instructions. This makes it two streams, but we could be accessing other streams as well. For instance, we could be accessing multiple arrays or multiple structures stored in memory in the same window of time. This is why consecutive accesses to the same page, or the same region, are somewhat infrequent. Hence, extremely low values of the stack distance are rarely seen. However, given that most programs have a substantial amount of temporal locality, we see a peak in the stack distance curve in the low to low-medium range of values – they are very frequent. This means that if an address is accessed, the probability that it will be accessed after k accesses to other addresses, is high if k is relatively small. Almost all computer systems take advantage of such a pattern because the stack distance curve roughly looks similar for cache accesses, page accesses, hard disk regions, etc.

The heavy tail arises because programs tend to make a lot of random accesses, tend to change phases and also tend to access a lot of infrequently used data. As a result, large stack distances are often seen. This explains the heavy tail in the representative plot shown in Figure 6.4. There are a lot of distributions that have heavy tails. Most of the time, researchers model this curve using the log-normal distribution. This is because it has a heavy tail as well as it is easy to analyze mathematically.

Let us understand the significance of the *stack distance*. It is a measure of temporal locality. Lower the average stack distance, higher the temporal locality. It basically means that we keep accessing the same pages over and over again in the same window of time. Similarly, higher the stack distance, lower the temporal locality. This means that we tend to re-access the same page after a long period of time. Such patterns are unlikely to benefit from standard architectural optimizations like caching. As discussed earlier, the log-normal
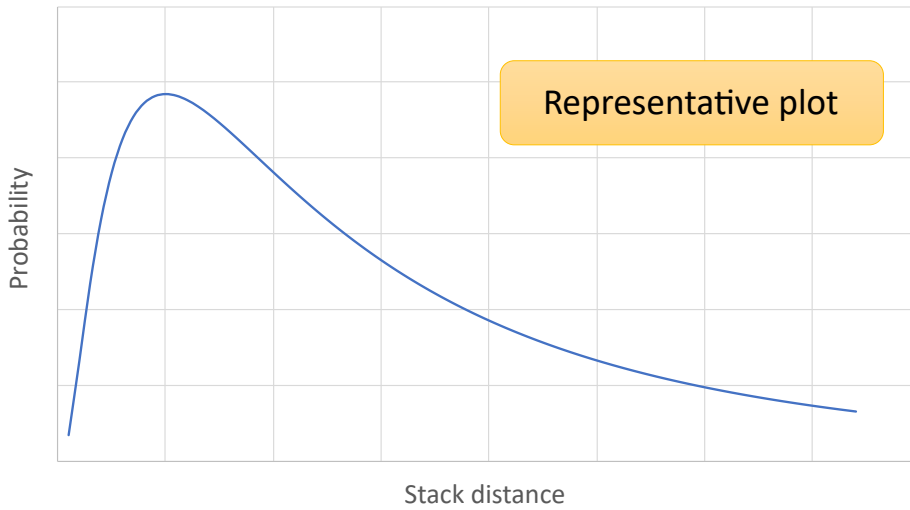
Figure 6.4: Representative plot of the stack distance

distribution is typically used to model the stack distance curve because it captures the fact that very low stack distances are rare, then there is a strong peak and finally there is a heavy tail. This is easy to interpret and also easy to use as a theoretical tool. Furthermore, we can use it to perform some straightforward mathematical analyses as well as also realize practical algorithms that rely on some form of caching or some other mechanism to leverage temporal locality.

Stack-based Algorithms

Let us now define the notion of a *stack-based algorithm*. It refers to a family of algorithms that use a conceptual stack to store pages. It is not a pure LIFO structure. However, it does have similarities with conventional stacks, hence, it is named that way. It is important to underscore the fact that an actual stack is not maintained.

The idea is to conceptually organize all the physical pages in memory as a stack. The page at the stack top has the least replacement priority. Whereas, the page at the bottom of the stack has the highest replacement priority. Assume that the size of the memory is n pages. Let $\mathcal{S}_n$ be the set of pages in a memory with n frames. The *stack property* states that $\mathcal{S}_n \subseteq \mathcal{S}_{n+1}$ for all values of $n \geq 1$. This means that if we add an additional frame, and create a larger memory, we are guaranteed to store all the pages that are stored in a smaller memory at any point of time. The assumption here is that the page access sequence is the same for both the memories.

> **Point 6.1.1**
>
> Let $\mathcal{S}_n$ be the set of pages in a memory with n frames. The *stack property* states that $\mathcal{S}_n \subseteq \mathcal{S}_{n+1}$ for all values of $n \geq 1$.

The stack property can easily be ensured if the two memories follow the stack

property at the beginning of the execution and adhere to some rules throughout the execution. Consider a replacement decision. Either both the memories make the same replacement decision or some page is evicted in $\mathcal{S}_{n+1}$ that is not present in $\mathcal{S}_n$. Consider two memories $\mathcal{M}_n$ and $\mathcal{M}_{n+1}$. Let the set of pages in $\mathcal{M}_n$ be S_n and the set of pages in $\mathcal{M}_{n+1}$ be S_{n+1}. Assume that the stack property holds till a certain point. Consider the first point at which the stack property fails to hold. Given that the stack property was followed up till now, it is not possible for the following to happen for the current access: there was a hit in S_n and miss in S_{n+1}. If there was a hit in both the memories, then there is no need for an eviction. Hence, this cannot be a point of divergence. This means that there must have been a miss in S_n. Let the current page accessed be p. Clearly, $p \notin S_n$. This means that page $q \in S_n$ had to be evicted to make space for p. Hence, we have the following relationship: $S_n = S_n^{old} + p - q$. Now, we know that $S_n \nsubseteq S_{n+1}$. This means that there was an eviction in S_{n+1}. Assume page r was evicted. $r \in S_n$ and $r \notin S_{n+1}$. This is where the stack property is being violated. By definition $r \neq q$. We also have $S_{n+1} = S_{n+1}^{old} + p - r$. Given that the stack property held up till now $q \in S_{n+1}$.

Let us look at the different choices made. $\mathcal{M}_n$ chose to retain r and evict q, whereas $\mathcal{M}_{n+1}$ chose to retain q and discard r. Assume some function $\mathcal{F}$ was used to determine the suitability of a page for eviction: higher the value, higher the need for eviction. For $\mathcal{M}_n$, $\mathcal{F}(r) < \mathcal{F}(q)$, and for $\mathcal{M}_{n+1}$, $\mathcal{F}(q) < \mathcal{F}(r)$. This means that the function $\mathcal{F}$ is dependent on the size of the memory. It is not global or universal.

As an example, consider the "least recently used" (LRU) algorithm. $\mathcal{F}$ is inversely proportional to the last-accessed time. The higher it is, higher the possibility of eviction. It is clear that this function is not dependent on the size of the memory. It is independent of the memory size. Hence, the stack property will not be violated. Let us refer to this function as the page cost function.

Optimal Page Replacement Algorithm

Similar to scheduling, we have an optimal algorithm for page replacement. Here the objective function is to minimize the number of page faults or conversely maximize the page hit rate in memory. The ideas are quite similar. Like the case with optimal scheduling, we need to make some hypothetical assumptions that are not realistic. Recall that we had done so in scheduling too, where we had assumed that we knew the exact execution duration of each job.

We start out with ordering all the pages in memory in ascending order of their "next use" time. This is how we organize our hypothetical stack. The moment a page is accessed, its new location is determined based on when it will be used next. Then for replacement, we choose that candidate page in memory that is going to be used or accessed the farthest in the future (bottom of the stack). The page cost function is the time of next access. This is independent of the memory size and thus the stack property is being followed.

It turns out that this algorithm is optimal. The proof technique is quite similar to what we had used to prove that a scheduling algorithm is optimal. We can use a contradiction-based technique and use exchange-based arguments to prove optimality, which in this case is the lowest page fault rate.

We shall see that there are many other algorithms that are also stack-based All of them have interesting properties in the sense that they avoid a certain kind

of anomalous behavior (discussed later when we discuss the FIFO replacement algorithm). It is easy to see that the page fault rate can never increase if we increase the size of the memory in a stack-based algorithm.

Least Recently Used (LRU) Algorithm

In the LRU algorithm, we conceptually tag each page in memory with the last time that it was accessed and choose that page for replacement that has the earliest access time. We assume that the past is a good predictor of the future – if a page has not been accessed in the recent past, then it is quite unlikely that it will be accessed in the near future.

This algorithm is stack-based. The priority is inversely proportional to the last-accessed time. Whenever we access a page, it is moved to the top of the stack. Recall that when we discussed stack distance, we had used such a scheme. It was nothing but an implementation of the Least Recently Used (LRU) replacement algorithm.

Let us now come to the issue of storing timestamps. We cannot add extra fields to main memory or the TLB to store additional timestamps – this will increase their storage overheads substantially and require hardware changes. We also cannot burden every memory access with computing and storing a timestamp. Furthermore, to find the least timestamp of a page in memory, we need to maintain either a stack or a priority queue. The stack maintains the relative order and thus needs to be updated on every access. A priority queue, on the other hand, requires $O(log(N))$ time for finding and updating entries. Hence, this scheme in its purest sense is impractical.

We can always store the timestamp in each page table entry. This will somewhat reduce the storage overheads in performance-critical structures like the TLB. The problem is that the page table is not accessed on every memory access. Hence, we will not be able to accurately maintain timestamps.

Hence, we need to make approximations such that this algorithm can be made practical. We don't want to set or compute bits on every memory access. Maintaining and updating LRU information needs to be an *infrequent operation*.

Practical LRU
Let us leverage the page protection bits that are a part of the page table as well as the TLB. These bits are needed to enforce access permissions. For example, if we are not allowed to write to a page, then its write access bit is set to 0. Our idea is to leverage these protection bits to add an estimate of the last access time to each page table entry. Let us start with marking all the pages as "not accessible". We set their access bits to 0. This idea may sound non-intuitive at the moment. But we will quickly see that this is one of the most efficient mechanisms of tracking page accesses and computing last-used information. Hence, this mechanism, even though it may sound convoluted, is actually quite popular and useful.

When we access a page, the hardware may find its access bit set to 0. This will lead to a page fault because of inadequate permissions. An exception handler will run, and it will figure out that the access bit was deliberately set to 0 such that an access can be tracked. Then it will set the access bit to 1 such that subsequent accesses go through seamlessly. However, the time at which the access bit was converted from 0 to 1 can be recorded, and this information can

be used to assist in the process of finding the LRU replacement candidate.

Given that the hardware does not provide any other way of efficiently recording the last-access information, this is the best that can be done. It is true that we are introducing deliberate page faults. However, these are soft page faults, where the page is there but the current task does not have the permission to access it. Such page faults are handled quickly. The kernel realizes that the access bits were deliberately set to 0 to track accesses. It sets them back to 1.

An astute reader may argue that over time all the access bits will get set to 1. This is correct, hence, there is a need to periodically reset all the access bits to 0. While finding a candidate for replacement, if an access bit is still 0, then it means that after it was reset the last time, the page has not been accessed. Therefore, we can conclude that this page has not been recently accessed and can possibly be replaced. This is a coarse-grained approach of tracking access information. It is however a very fast algorithm and does not burden every memory access.

We can do something slightly smarter subject to the computational bandwidth that we have. We can look at the timestamp stored along with each page table entry. If the access bit is equal to 0, then we can look at the timestamp. Recall that the timestamp corresponds to the time when the page's access bit last transitioned from 0 to 1. This means that the page was accessed and there was a soft page fault. Later on this access bit was again reset to 0 because of periodic clearing. We can use that timestamp as a proxy for the recency of the page access. Smaller the timestamp, higher the eviction probability. This approximate scheme may look appealing, however, in practice its accuracy is questionable and thus is not used in real-world implementations. Instead, the WS-Clock family of approximations of the LRU scheme are used.

WS-Clock Algorithm
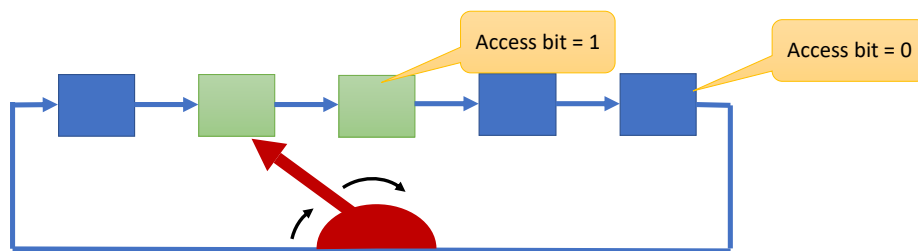
Figure 6.5: The WS-Clock algorithm

Let us now propose a more practical approximation of the LRU algorithm. The WS-Clock page replacement algorithm is shown in Figure 6.5. Here WS stands for "working set", which we shall discuss later in Section 6.1.3.

Every physical page in memory is associated with an access bit. It is stored in the corresponding page table entry. A pointer like the minute hand of a clock points to a physical page; it is meant to move through all the physical pages one after the other (in the list of pages) until it wraps around.

If the access bit of the page pointed to by the pointer is equal to 1 (recently used), then it is set to 0 (unused) when the pointer traverses it. There is no need

to periodically scan all the pages and set their access bits to 0. This will take a lot of time. Instead, in this algorithm, once there is a need for replacement we traverse the list of physical pages from the last position. We check the access bit and if it is set to 1, we reset it to 0. This means that if the page is recently used, we mark it as unused. However, if the access bit is equal to 0, then we select that page for replacement. For the time being, the pointer stops at that page. Next time the pointer starts from the same page and keeps traversing the list of pages towards the end until it wraps around at the end.

This algorithm can approximately find the pages that are not recently used and select one of them for eviction. It turns out that we can do better if we differentiate between unmodified and modified pages in systems where the swap space is inclusive – every page in memory has a copy in the swap space, which could possibly be stale. The swap space in this case acts as a lower-level cache.

WS-Clock Second Chance Algorithm

Let us now look at a slightly improved version of the algorithm that uses 2-bit state subject to the caveats that we have mentioned. The 2 bits are the *access bit* and the *modified bit*. The latter bit is set when we write to a word in the page. The corresponding state table is shown in Table 6.1.

⟨ **Access bit, Modified bit** ⟩	**New State**	**Action**
$\langle 0,0 \rangle$	$\langle 0,0 \rangle$	Go ahead and replace
$\langle 0,1 \rangle$	$\langle 0,0 \rangle$	Schedule a write-back, move forward.
$\langle 1,0 \rangle$	$\langle 0,0 \rangle$	Move forward
$\langle 1,1 \rangle$	$\langle 1,0 \rangle$	Frequently used frame; move forward. Schedule a write-back.

Table 6.1: State-action table in the WS-Clock second chance algorithm

If both the bits are equal to 0, then they remain so, and we go ahead and select that page as a candidate for replacement. On the other hand if they are equal to $\langle 0,1 \rangle$, which means that the page has been modified and after that its access bit has been set to 0, then we perform a write-back and move forward. The final state in this case is set to $\langle 00 \rangle$ because the data is not deemed to be modified anymore since it is written back to memory. Note that every modified page in this case has to be written back to the swap space whereas unmodified pages can be seamlessly evicted given that the swap space has a copy. As a result, unmodified pages are prioritized for eviction.

Next, let us consider the combination $\langle 1,0 \rangle$. Here, the access bit is 1, so we set it to 0. The resulting combination of bits is now $\langle 0,0 \rangle$; we move forward.

Finally, if the combination of these 2 bits is $\langle 1,1 \rangle$, then we perform the write-back, and reset the new state to $\langle 1,0 \rangle$. This means that it is clearly a frequently used page that gets written to, and thus it should not be evicted or downgraded – the access bit should not be set to 0. It deserves a second chance.

This is per se a simple algorithm, which takes the differing overheads of reads and writes into account. For writes, it gives a page a second chance in a certain sense.

We need to understand that such LRU-approximating algorithms are quite heavy. They introduce artificial page access faults. Of course, they are not as onerous as full-blown page faults because they do not fetch data from the underlying storage device that takes millions of cycles. Here, we only need to perform some bookkeeping and change the page access permissions. This is much faster than fetching the entire page from the hard disk or NVM drive. Such soft page faults still lead to an exception and require time to service. There is some degree of complexity involved in this mechanism. But at least we are able to approximate LRU to some extent.

FIFO Algorithm

The queue-based FIFO (first-in first-out) algorithm is one of the most popular algorithms in this space, and it is quite easy to implement because it does not require any last-usage tracking or access bit tracking. It is easy to implement primarily because all that we need to do is that we need to have a simple priority queue in memory that stores all the physical pages based on when they were brought into memory. The page that was brought in the earliest is the replacement candidate. There is no run time overhead in maintaining or updating this information. We do not spend any time in setting and resetting access bits or in servicing page access faults. Note that this algorithm is not stack based, and it does not follow the *stack property*. The cost function is the reciprocal of the time at which the physical page was last added to the memory. This is clearly dependent on the size of the memory. For memories of different sizes, pages can keep getting evicted and coming in. The last time it entered the memory is not a global metric. This violates the stack priority. This is not a good thing as we shall see shortly.

Even though this algorithm is simple, it suffers from a very interesting anomaly known as the Belady's Anomaly [Belady et al., 1969] owing to the fact that the stack property is not followed. Let us understand it better by looking at the two examples shown in Figures6.6 and 6.7. In Figure 6.6, we show an access sequence of physical page ids (shown in square boxes). The memory can fit only four frames. If there is a page fault, we mark the entry with a cross otherwise we mark the box corresponding to the access with a tick. The numbers at the bottom represent the contents of the FIFO queue after considering the current access. After each access, the FIFO queue is updated.

If the memory is full, then one of the physical pages (frames) in memory needs to be removed. It is the page that is at the head of the FIFO queue – the earliest page that was brought into memory. The reader should take some time and understand how this algorithm works and mentally simulate it. She needs to understand and appreciate how the FIFO information is maintained and why this algorithm is not stack based.

In this particular example shown in Figure 6.6, we see that we have a total of 10 page faults. Surprisingly, if we reduce the number of physical frames in memory to 3 (see Figure 6.7), we have a very counter-intuitive result. We would ideally expect the number of page faults to increase because the memory size is smaller. However, we observe an *anomalous* result. We have 9 page faults (one page fault less than the larger memory with 4 frames) !!!

The reader needs to go through this example in great detail. She needs to understand the reasons behind this anomaly. These anomalies are only seen in
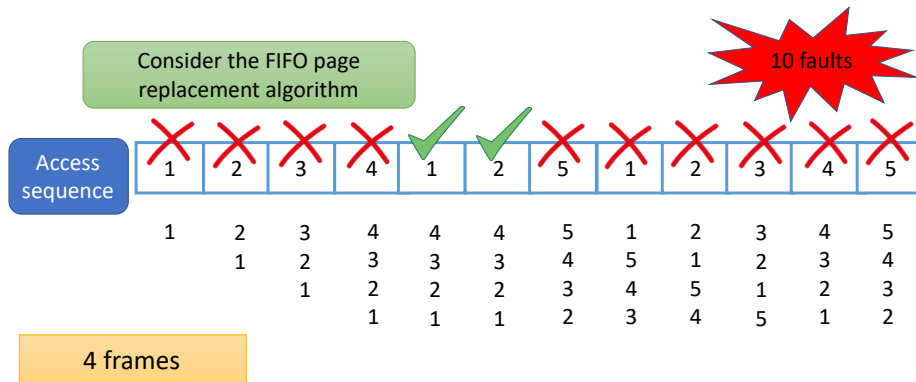
Figure 6.6: FIFO algorithm with memory capacity equal to 4 frames
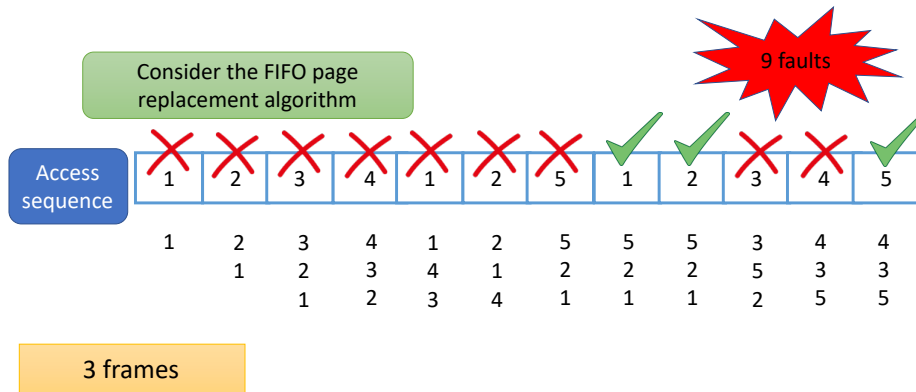
Figure 6.7: FIFO algorithm with memory capacity equal to 3 frames

algorithms that are not stack-based. Recall that in a stack-based algorithm, we have the stack property – at all points of time the set of pages in a larger memory is a superset of the pages that we would have in a smaller memory. Hence, we cannot observe such an anomaly. Now, we may be tempted to believe that this anomaly is actually limited to small discrepancies. This means that if we reduce the size of the memory, maybe the size of the anomaly is quite small (limited to a very few pages).

However, this presumption is sadly not true. It was shown in a classic paper by Fornai et al. [Fornai and Iványi, 2010a, Fornai and Iványi, 2010b] that a sequence always exists that can make the discrepancy arbitrarily large. In other words, it is unbounded. This is why the Belady's anomaly renders many of these non-stack-based algorithms completely ineffective. They perform very badly in the worst case. One may argue that such "bad" cases are pathological and rare. But in reality, such bad cases do occur to a limited extent. This significantly reduces the performance of the system because page faults are associated with massive overheads.

> **Point 6.1.2**
>
> Let us now summarize our discussion. A pure stack-based algorithm does
> not suffer from the Belady's anomaly. In line with this philosophy, we
> introduced the WS-Clock and the WS-Clock Second Chance algorithms
> that approximate LRU. The FIFO replacement algorithm is in compari-
> son much easier to implement. However, it exhibits the classic Belady's
> anomaly. The worst case performance of FIFO can be arbitrarily low.

6.1.3 The Notion of the Working Set

Let us now come to the notion of a "working set". Loosely speaking, it is the set
of pages that a program accesses in a short duration or small window of time.
It pretty much keeps on repeatedly accessing pages within the working set. In
a sense, it remains confined to all the pages within the working set in a *small*
window of time. Of course, this is an informal definition. Proposing a formal
definition is somewhat difficult because we need to quantify how short a time
duration we need to consider.
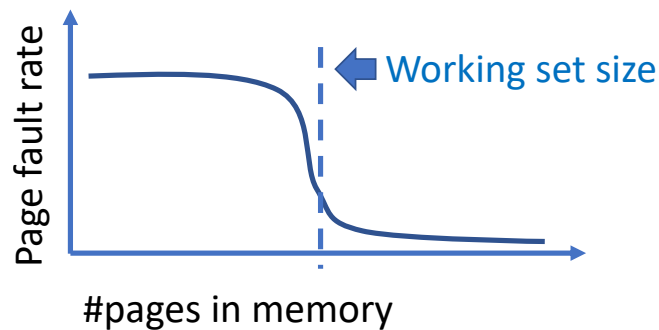
Figure 6.8: Page fault rate versus the working set size

There is a different way of answering this question. It is scientifically more
reasonable. Consider the graph shown in Figure 6.8. The x axis is the number
of pages that we have in memory and the y axis is the page fault rate. We
typically observe that the page fault rate is initially very high. It continues to
reduce very sluggishly until a certain point and after that there is a sudden dip
– the page fault rate reduces substantially. It continues to be low beyond this
point of sharp reduction. More or less all real-world programs show a similar
behavior even though the shape of the curve tends to vary across them.

We can define the point of sharp page fault reduction as the working set size
of the program. If we have more pages than this threshold (working set), the
page fault rate will be low. Otherwise, it will be very high. Note that this is
typical behavior that has been observed empirically.

The notion of the working set can be construed as the set of pages that a
program tends to access repeatedly within a small window of time. It suffers
a lot in terms of performance if the size of the memory that is allocated to it
in terms of the number of pages is less than the size of the working set. Even

though Figure 6.8 is a representative figure, it is widely accepted that almost all real-world programs show a similar behavior and that is why the notion of the working set is reasonably well-defined using such arguments. The slope of the line in the vicinity of the working set size can be steep for some benchmarks and can be relatively less steep for others, however, this effect is nonetheless always visible to some extent.

6.1.4 Shared- Memory-Based Inter-Process Communication

Example 6.1.1

Create a shared memory channel between two processes. Send a message from a writer process to a reader process.

Answer: Here is the code for the writer process.

Listing 6.1: Writer process

```c
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#include <unistd.h>

#define SHM_SIZE 1024  // shared memory size (in bytes)

int main() {
    // create a unique key for the shared mem. segment
    key_t key = ftok("shmfile", 59);

    // Create the segment and get a handle to it
    int shmid = shmget(key, SHM_SIZE, 0666|IPC_CREAT);

    // attach the segment to a string pointer
    char *str = (char *) shmat(shmid, NULL, 0);

    // Write to the shared memory segment
    strcpy(str, "Message from the writer.");
    printf("Writer wrote: %s\n", str);

    shmdt(str);  // detach from shared memory

    return 0;
}
```

After running the writer, it is necessary to run the reader process.

Listing 6.2: Reader process

```c
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
```

```
#define SHM_SIZE 1024

int main() {
    // Use the same unique key as the writer
    key_t key = ftok("shmfile", 59);

    // Get the segment corresponding to the key
    int shmid = shmget(key, SHM_SIZE, 0666);

    // attach the shared memory segment
    char *str = (char *) shmat(shmid, NULL, 0);

    // Read the shared memory segment
    printf("Reader read: %s\n", str);

    shmdt(str);  // detach from shared memory

    // destroy the shared memory segment
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

The `ftok` library call takes the name of a file and an integer id. The key is a unique combination of these two arguments. This is done such that different processes can generate the same key, as long as the "id" is the same.

Next, there is a need to create the shared memory segment, if the process is the first writer. Typical read/write permissions are assigned in our example to the user, group and the world (more in Chapter 8). Next, the reader and writer need to attach a string pointer with the shared memory region. This creates the relevant mappings in the page table. The virtual address corresponding to the string variable `str` points to the shared memory segment. This further means that any read and write operation on bytes within the string happens directly on shared memory. The other process can immediately see the values written by write operations because of the way the physical page is mapped in the virtual address space of both processes (reader and writer).

In our example, the writer uses the `strcpy` function and the reader reads the `str` pointer directly. Any read/write operation can be used. Because of the way the mapping is created, any time a virtual address in the array `str` is accessed, the access will directly go to the shared physical page. This mechanism can thus allow many processes to access the same physical page concurrently.

After the shared memory operations have finished there is a need to detach the segment. This does the reverse. It unmaps the shared memory segment. The corresponding virtual address does not point to the shared page anymore. The reader process in our example, subsequently destroys the shared memory segment such that it is not usable anymore.

6.2   Virtual and Physical Address Spaces

The most important concepts in this space are the design of the overall virtual
memory space, the page table and associated structures. We will begin with
a short discussion on Linux page tables and then move on to discuss the way
in which metadata associated with a single physical page is stored (in `struct
page`). The kernel also has the notion of *folios*, which are basically of set of
pages that have contiguous addresses in both the physical and virtual address
spaces. They are a recent addition to the kernel (v5.18) and are expected to
grow in terms of popularity, usage and importance.

6.2.1   The Virtual Memory Map
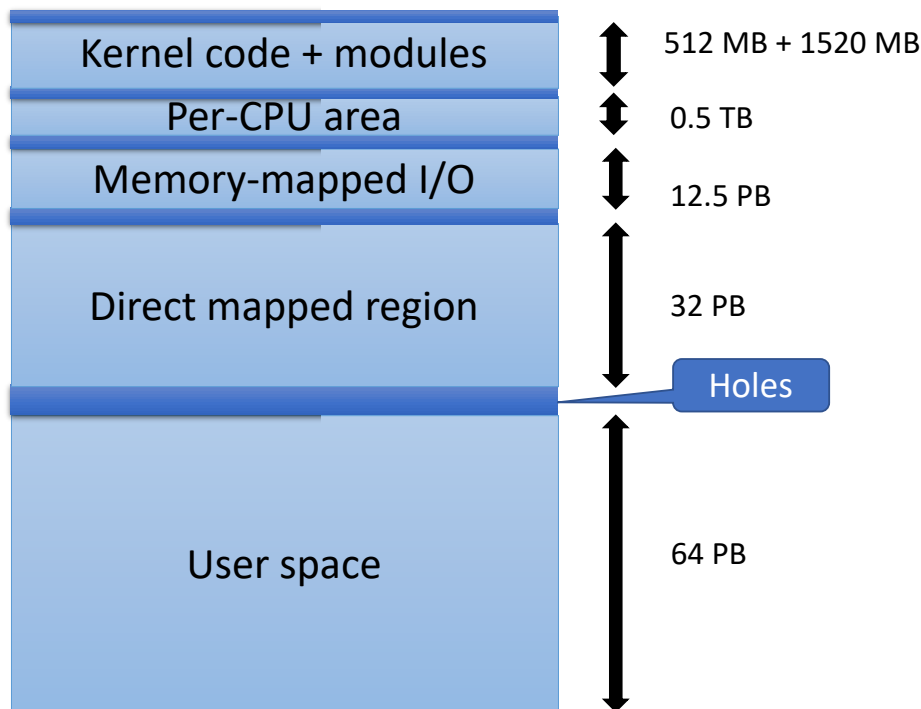
Figure 6.9: The overall virtual memory map (kernel + user)
source : Documentation/x86/x86_64/mm.rst

Let us first understand the overall virtual memory map (user + kernel). In
Linux, the virtual address space is partitioned between all the kernel threads
and a user process. There is no overlap between the user and kernel virtual
address spaces because they have to be strictly separated. Let us consider the
case of a 57-bit virtual addressing system. The total virtual memory size is 128
PB (2^{57} bytes). We partition the virtual memory space into two parts: 64 PB
for the user process and 64 PB for kernel threads.

The user space virtual memory is further partitioned into different sections
such as the text, stack and heap (see Section 2.2). In this chapter, let us look

at the way the kernel virtual memory is partitioned (refer to Figure 6.9). Note that the figure is not drawn to scale – we have only shown some important regions. The data shown in the figure is by no means exhaustive. Refer to the documentation (cited in the figure's caption) for a more detailed list of kernel memory regions.

Note the 32 PB direct-mapped region. In this region, the virtual and physical addresses are either the same or are linearly related (depending upon the version and architecture), which basically means that we can access physical memory *directly*. This is always required because the kernel needs to work with real physical addresses sometime, especially while dealing with external entities such as I/O devices, the DMA controller and the booting subsystem. It is also used to store the page tables. Page tables should not be stored in virtual memory because we can have page faults while accessing them. Hence, they need to be accessed directly.

This memory is also useful whenever we want to create a large set of buffers that are shared with I/O devices, or we want to create a cache of structures of a known size. Essentially, this entire region can be used for any custom purpose especially when contiguity of physical memory addresses is required and page faults are not allowed.

Next, we have a memory-mapped I/O region that stores all the pages that are mapped to I/O devices for the purpose of memory-mapped I/O. Another important region in the kernel's virtual memory address space is the per-CPU area, which we have seen to play a very important role in storing information related to the `current task`, part of the context and preemption-related flags.

The core kernel code per se is quite small. We only reserve 512 MB for the kernel code. It is important to note that a large part of the overall kernel code comprises driver code. This code is loaded on demand based on the devices that are plugged to the machine. All of this code is actually present in modules (1520 MB reserved in the virtual address space) that are loaded or unloaded dynamically. Modules used to have more or less unfettered access to the kernel's data structures, however off late this is changing.

In general, modules are typically used to implement device drivers, file systems, and cryptographic protocols/mechanisms. They help keep the core kernel code small, modular and clean. Of course, security is a big concern while loading kernel modules and thus module-specific safeguards are increasingly getting more sophisticated – they ensure that modules have limited access to only the functionalities that they need. With novel module signing methods, we can ensure that only *trusted modules* are loaded. 1520 MB is a representative figure for the size reserved for storing module-related code and data in kernel v6.2. Note that this is not a standardized number, it can vary across Linux versions and is also configurable.

6.2.2 The Page Table

Figure 6.10 shows the `mm_struct` structure that we have seen before. It specifically highlights a single field, which stores the page table (`pgd_t *pgd`). The page table is also known as the page directory in Linux. There are two virtual memory address sizes that are commonly supported: 48 bits and 57 bits. We have chosen to describe the 57-bit address in Figure 6.10. We observe that there are five levels in a page table. The highest level of the page table is known as the
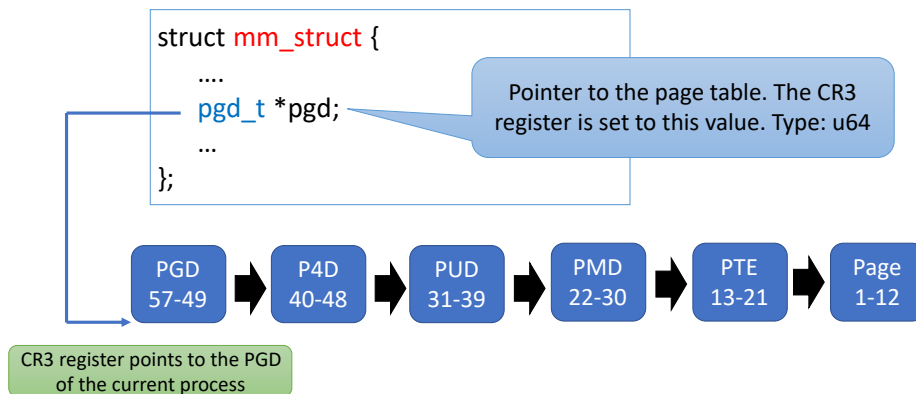
Figure 6.10: The high-level organization of the page table (57-bit address)

page directory (PGD). Its starting address is stored in the CR3 MSR (model specific register). CR3 stores the starting address of the page table (highest level) and is specific to a given process. This means that when the process changes, the contents of the CR3 register also need to change. It needs to point to the page table of the new process. There is a need to also flush the TLB if the contents of the CR3 register change. This is very expensive. Hence, various kinds of optimizations have been proposed, which we shall discuss later.

We have already seen in Chapter 3 that the contents of the CR3 register do not change when we make a process-to-kernel transition or in some cases in a kernel-to-process transition as well. Here the term *process* refers to a user process. The main reason for this is that changing the virtual memory context is associated with a lot of performance overheads and thus there is a need to minimize such events as much as possible, and all kernel threads share their virtual address space.

The page directory is indexed using the top 9 bits of the virtual address (bits 49-57). Then we have four more levels. For each level, the next 9 bits (towards the LSB) are used to address the corresponding table. The reason that we have a five-level page table here is because we have 57 virtual address bits and thus there is a need to have more page table levels. Our aim is to reduce the memory footprint of page tables as much as possible and properly leverage the sparsity in the virtual address space. The details of all of these tables are shown in Table 6.2. We observe that the leaf-level entry is the *page table entry*, which contains the mapping between the virtual page number and the page frame number (or the number of the physical page) along with some page protection information.

Page Table Entry

Each page table entry also contains permission bits (see Table 6.3). This information is also kept in the TLB such that the hardware can check if a given operation is allowed or not without consulting the page table. For example, if we are not allowed to execute code within the page, then execute permissions will not be given. This is very important from a security perspective. Similarly,

Acronym	Full form
PGD	Page Global Directory
P4D	Fourth level page table
PUD	Page Upper Directory
PMD	Page Middle Directory
PTE	Page Table Entry

Table 6.2: All the constituent tables of a 5-level page table
source : arch/x86/include/asm/pgtable_types.h

for code pages, write access is typically turned off – this ensures that viruses or malware cannot modify the code of the program and cannot hijack the control flow. Finally, we also need a bit to indicate whether the page can be accessed or not. Recall that we had used such bits to track the usage information for the purposes of LRU-based replacement. They can be used to induce soft page faults.

Acronym	Full form
PROT_READ	Read permission
PROT_WRITE	Write permission
PROT_EXEC	Execute permission
PROT_SEM	Can be used for atomic ops
PROT_NONE	Page cannot be accessed

Table 6.3: Page protection bits (pgprot_t)
source : include/uapi/asm − generic/mman − common.h

Walking the Page Table

Listing 6.3: The follow_pte function (assume the entry exists)
source : mm/memory.c#L5350

```c
int follow_pte(struct mm_struct *mm, unsigned long address,
        pte_t **ptepp, spinlock_t **ptlp) {
    pgd_t *pgd;
    p4d_t *p4d;
    pud_t *pud;
    pmd_t *pmd;
    pte_t *ptep;

    pgd = pgd_offset(mm, address);
    p4d = p4d_offset(pgd, address);
    pud = pud_offset(p4d, address);
    pmd = pmd_offset(pud, address);
    ptep = pte_offset_map_lock(mm, pmd, address, ptlp);

    *ptepp = ptep;
    return 0;
```

```
}
```

Listing 6.3 shows the code for traversing the page table (`follow_pte` function) assuming that an entry exists. We first walk the top-level page directory, and find a pointer to the next level table. Next, we traverse this table, find a pointer to the next level, so on and so forth. Finally, we find the pointer to the page table entry. However, in this case, we also pass a pointer to a spinlock. The page table entry is locked prior to returning a pointer to it. This allows us to make changes to the page table entry. It needs to be subsequently unlocked after it has been accessed.

Let us now delve slightly deeper into the code that traverses the PMD table. A representative example for traversing the PMD table is shown in Listing 6.4. We will start with a PUD table entry that contains a pointer to a PMD table. We need to find the index of the PMD entry using the function `pmd_index` and add it to the base address of the PMD table. This gives us a pointer to an entry in the PMD table.

Listing 6.4: Accessing the page table at the PMD level
source : include/linux/pgtable.h#L109

```
/* include/linux/pgtable.h */
pmd_t *pmd_offset(pud_t *pud, unsigned long address) {
    return pud_pgtable(*pud) + pmd_index(address);
}

/* right shift by 21 positions and AND with 511 (512-1) */
unsigned long pmd_index(unsigned long address) {
    return (address >> PMD_SHIFT) & (PTRS_PER_PMD - 1);
}

/* arch/x86/include/asm/pgtable.h */
/* Align the address to a page boundary (only keep the bits
   in the range 13-52),
 add this to PAGE_OFFSET and return */
pmd_t *pud_pgtable(pud_t pud) {
    return (pmd_t *) __va(pud_val(pud) & pud_pfn_mask(pud));
}

/* arch/x86/include/asm/page.h */
/* virtual address = physical address + PAGE_OFFSET (start
   of direct-mapped memory) */
#define __va(x) ((void *)((unsigned long)(x)+PAGE_OFFSET))
```

First consider the `pmd_index` inline function that takes the virtual address as input. We need to next extract bits 22-30. This is achieved by shifting the address to the right by 21 positions and then extracting the bottom 9 bits (using a bitwise AND operation). The function returns the entry number in the PMD table. This is multiplied with the size of a PMD entry and then added to the base address of the PMD page table that is obtained using the `pud_pgtable` function.

Let us now look at the `pud_pgtable` function. It relies on the `__va` inline function that takes a physical address as input and returns the virtual address. The reverse is done by the `__pa` inline function (or macro). In `__va(x)`, we

simply add the argument `x` to an address called `PAGE_OFFSET`. This is not the offset within a page, as the name may suggest. It is an offset into a memory region where the page table entries are stored. These entries are stored in the direct-mapped region of kernel memory. The `PAGE_OFFSET` variable points to some point within this region (depending upon the architecture). We are realizing a linear conversion between a physical and virtual address.

The inline `pud_pgtable` function invokes the `__va` function with an argument that is constructed as follows. `pud_valpud` returns the bits corresponding to the physical address of the PMD table. We compute a bitwise AND between this value and a constant that has all 1s between bit positions 13 and 52 (rest 0s). The reason is that the maximum physical address size is assumed to be 2^{52} bytes in Linux. Furthermore, we are aligning the address with a page boundary, hence, the first 12 bits (offset within the page) are set to 0. The last reason is that in the physical address stored in each entry, bits 1-12 are used to store some other metadata information as well. All of this needs to be removed prior to the access.

This physical address is then converted to a virtual address using the `__va` function. The output of the `pud_pgtable` function thus returns the virtual address of the starting address of the PMD page table. We then add the offset of the PMD entry to this address and find the virtual address of the PMD entry.

6.2.3 Pages and Folios

Let us now discuss pages and folios in more detail. For every physical page (frame), we store a data structure called the page structure (`struct page`). It is important to store some metadata in this structure such as the nature of the page, whether it is anonymous or not, whether it is a memory-mapped page and if it is usable for DMA operations. Note that a page in memory can actually represent many kinds of data: regular data, I/O data, atomic variables, etc. We thus need an elaborate page structure.

As discussed earlier, a *folio* is a set of pages that has contiguous addresses in both the physical and virtual address spaces. They can reduce the translation overhead significantly and make it easier to interface with I/O devices and DMA controllers.

struct page

`struct page` is defined in include/linux/mm_types.h. It is a fairly complex data structure that extensively relies on unions. Recall that a union in C is a data type that can store multiple types of data in the same memory location. It is a good data type to use if we want it to store many types of data, where only one type is used at a time.

The page structure begins with a set of flags that indicate the status of the page. They indicate whether the page is locked, modified, in the process of being written back, active, already referenced or reserved for special purposes. Then there is a union whose size can vary from 20 to 40 bytes depending upon the configuration. We can store a bunch of things such as a pointer to the address space (in the case of I/O devices), a pointer to a pool of pages, or a page map (to map DMA pages or pages linked to an I/O device). Then we have a reference count, which indicates the number of entities that are currently holding

a reference of the page. This includes regular processes, kernel components or even external devices such as DMA controllers.

We need to ensure that before a page is recycled (returned to the pool of pages), when its reference count is equal to zero. It is important to note that the `page` structure is ubiquitously used and that too for numerous purposes, hence it needs to have a very flexible structure. This is where using a union with a large number of options for storing diverse types of data turns out to be very useful.

Folios

Let us now discuss folios [Corbet, 2022, Corbet, 2021]. A folio is a generalization of a page. It is a *compound* or *aggregate* page that contains multiple pages. The number of pages needs to be a power of 2. The pages are contiguously allocated in both the physical and virtual memory spaces. The reason that folios were introduced is because memories are very large as of today, and it is very difficult to handle the millions of pages that they contain. The sheer translation overhead and overhead for maintaining page-related metadata and information is quite prohibitive. Hence, a need was felt to group consecutive pages into larger units called folios. Specifically, a folio points to the first page in a group of pages (compound page). Additionally, it stores the number of pages that are a part of it.

The earliest avatars of folios were meant to be a contiguous set of virtual pages, where the folio per se was identified by a pointer to the head page (first page). It was a single entity insofar as the rest of the kernel code was concerned. This in itself is a very useful concept because we are grouping contiguous virtual memory pages based on some notion of application-level access patterns.

If the first page of the folio is accessed, then in all likelihood the rest of the pages will also be accessed very soon given that modern programs have a lot of spatial locality. Hence, it makes a lot of sense to prefetch these pages to memory in anticipation of them being used in the near future. However, over the years the thinking has somewhat changed even though folios are still in the process of being fully integrated into the kernel. Now most interpretations try to also achieve contiguity in the physical address space as well. This has a lot of advantages with respect to I/O, DMA accesses and reduced translation overheads. Let us discuss another angle.

Almost all server-class machines as of today have support for *huge pages*, which have sizes ranging from 2 MB (21 address bits) to 1 GB (30 address bits) [1]. They reduce the pressure on the TLB and page tables, and also increase the TLB hit rate as well. We maintain a single entry for the entire huge page. Consider a 1 GB huge page. It can store 2^{18} 4 KB pages. If we store a single mapping for it, then we are basically reducing the number of entries that we need to have in the TLB and page table substantially. Of course, this requires hardware support and also may sometimes be perceived to be wasteful in terms of memory. It can lead to internal fragmentation. However, in today's day and age we have a lot of physical memory. For many applications this is a very useful facility and the entire 1 GB region can be represented by a set of folios – this simplifies its management significantly.

---

[1] Please relate the numbers 21 and 30 to the structure of the page table. It will be clear why these numbers were chosen. They naturally fall at table boundaries.

Furthermore, I/O and DMA devices do not use address translation. They need to access physical memory directly, and thus they benefit by having a large amount of physical memory allocated to them. It becomes very easy to transfer a huge amount of data directly to/from physical memory if they have a large contiguous allocation. Additionally, from the point of view of software, it also becomes much easier to interface with I/O devices and DMA controllers because this entire memory region can be mapped to a folio. The concept of a folio along with a concomitant hardware mechanism such as huge pages enables us to perform such optimizations quite easily. We thus see that the folio as a multifaceted mechanism that enables prefetching and efficient management of I/O and DMA device spaces.

Given that a folio is perceived to be a single entity, all usage and replacement-related information (LRU stats) are maintained at the folio level. It basically acts like a single page. It has its own permission bits as well as copy-on-write status. Whenever a process is forked, the entire folio acts as a single unit like a page and is copied in totality when there is a write to any constituent page. LRU information and references are also tracked at the folio level.

Mapping the `struct page` to the Page Frame Number (and vice versa)

Let us now discuss how to map a page or folio structure to a page frame number (pfn). There are several simple mapping mechanisms. Listing 6.5 shows the code for extracting the pfn from a page table entry (`pte_pfn` macro). We simply right shift the address by 12 positions (`PAGE_SHIFT`).

Listing 6.5: Converting the page frame number to the `struct page` and vice versa
source : include/asm − generic/memory_model.h#L39

```
#define pte_pfn(x) phys_to_pfn(x.pte)
#define phys_to_pfn(p) ((p) >> PAGE_SHIFT)

#define __pfn_to_page(pfn)                  \
({ unsigned long __pfn = (pfn);             \
    struct mem_section *__sec = __pfn_to_section(__pfn); \
    __section_mem_map_addr(__sec) + __pfn;              \
})
```

The next macro `__pfn_to_page` has several variants. A simpler avatar of this macro simply assumes a linear array of `page` structures. There are n such structures, where n is the number of frames in memory. The code in Listing 6.5 shows a more complex variant where we divide this array into a bunch of *sections*. We figure out the section number from the pfn (page frame number), and every section has a section-specific array. We find the base address of this array and add the page frame number to it to find the starting address of the corresponding `struct page`. The need for having sections will be discussed when we introduce *zones* in physical memory (in Section 6.2.5).

Point 6.2.1

It may appear that if there are N frames in memory, then each section needs to store N `struct pages`. This is not true. It

would be wasteful in terms of space. Let us thus explain Linux's sparse memory mapping mechanism that uses the notion of *biased pointers*. The function `__section_mem_map_addr` does not exactly return the base address of the section. Let us assume that it returns address A (unit is `struct page *`). We actually return $A = base\_address - starting\_page\_frame\_number$. For example, if $starting\_page\_frame\_number$ is 32 and the pfn is 36, then we are actually adding 4 to the base address of the section.

6.2.4 Managing the TLB

TLB Design

Let us now look at TLB-specific optimizations. Note that it is important to manage the TLB well primarily because TLB misses are expensive. We need to perform expensive page walks either in software or hardware. In either case, the overheads are quite high. This is why in modern processors, the TLB is a heavily optimized structure and a lot of effort is spent in minimizing TLB misses. The TLB also lies on the critical path of address translation and is thus a very latency-sensitive component. Hence, it is necessary to create a scheme that leverages both software and hardware to efficiently manage the TLB.

A TLB is designed like a cache (typically with 4 to 16-way associativity). A modern TLB has multiple levels: an i-TLB for instructions, a d-TLB for data and then a shared L2 TLB. In some processors it is possible to configure the associativity, however in most high-performance implementations, the associativity cannot be modified. Each entry of the TLB corresponds to a virtual page number; it stores the number of the physical frame/page and also contains some metadata that includes the page protection bits.

Let us consider a baseline implementation. The TLB maintains the mappings corresponding to a virtual address space. This is why when we load a new user process, we change the virtual address space by changing the base address of the page table that is stored in the CR3 register. There is also a need to flush the TLB because the entries in the TLB now correspond to the previous user process. This is very expensive because this increases the TLB miss rate significantly for both the new process as well as for the process that is being swapped out (when it runs again).

There is clearly a need to optimize this process such that we do not have to flush the entire TLB – we can minimize the number of misses. TLBs in Intel processors already provide features where we can mark some entries as global(G) and ensure that they are not flushed. For instance, the entries corresponding to the kernel's virtual address space can be marked as global – they will then not get flushed. Recall that the virtual address space is partitioned between the user address space and the kernel address space based on the value of the MSB bit (48th or 57th bit). The kernel's address space remains the same across user processes and thus there is no need to flush its entries from the TLB. Keeping such concerns in mind, Intel provides the `invlpg` instruction that can be used to selectively invalidate entries in the TLB without flushing all of it. This is clearly one positive step in effective TLB management – only flush those entries

that will either cause a correctness problem or will not be required in the near future.

We can do more. The main reason for flushing the TLB in whole or in part is because a mapping may not remain valid once we change the user process. By splitting the virtual address space between user processes and the kernel, we were able to avoid TLB flushes when we switch to the kernel primarily because we use a different non-overlapping set of virtual addresses. Hence, we can maintain the mappings of the user process that got interrupted – there is no issue. Again while exiting the kernel, if we are returning to the same user process, which is most often the case, then also there is no need to flush the TLB because we are not loading a new virtual address space. The mappings that were already there in the TLB can be reused. Note that there is a possibility that the kernel may have evicted some mappings of the user process, however we expect a lot of them to be still there, and they can be reused. This will consequently reduce the TLB miss rate once the user process starts to run again. This is the main advantage that we gain by partitioning the overall 48 or 57-bit virtual address space – it avoids costly TLB flushes (one while moving from the user process to the kernel and while switching back).

Now assume the more general case where we are switching to a new user process. In this case, the existing mappings for the user process that is being swapped out cannot be reused, and they have to be removed. It turns out that we can do something intelligent here ☺. If we can annotate each TLB entry with the process ID, then we do not have to flush the TLB. Instead, we can make the process ID a part of the memory access and use only those mappings in the TLB that belong to the current process. This is breaking a key abstraction in OS design – we are blurring the separating line between software and hardware. We have always viewed process IDs as pure software-level concepts, but now we want to make them visible to the hardware. We are breaking the long-held abstraction that software and hardware should be as independent of each other as possible. However, if we do not do this, then our TLB miss rates will be very high because every time the user process changes, its entries have to be flushed from the TLB. Hence, we need to find some kind of middle ground here.

ASIDs

Intel x86 processors have the notion of the processor context ID (PCID), which in software parlance is also known as the address space ID (ASID). We can take some important user-level processes that are running on a CPU and assign them a PCID each. Then their corresponding TLB entries will be tagged/annotated with the PCID. Furthermore, every memory access will now be annotated with the PCID (conceptually). Only those TLB entries will be considered that match the given PCID. Intel CPUs typically provide 2^{12} (=4096) PCIDs. One of them is reserved, hence practically 4095 PCIDs can be supported. There is no separate register for it. Instead, the top 12 bits of the CR3 register are used to store the current PCID.

Now let us come to the Linux kernel. It supports the generic notion of ASIDs (address space IDs), which are meant to be architecture independent. Note that it is possible that an architecture does not even provide ASIDs.

In the specific case of Intel x86-64 architectures, an ASID is the same as a PCID. This is how we align a software concept (ASID) with a hardware concept

(PCID). Given that the Linux kernel needs to run on a variety of machines and all of them may not have support for so many PCIDs, it needs to be slightly more conservative, and it needs to find a common denominator across all the architectures that it is meant to run on. For the current kernel (v6.2), the developers decided support only 6 ASIDs, which they deemed to be enough. This means that out of 4095, only 6 PCIDs on an Intel CPU are used. From a performance perspective, the kernel developers found this to be a reasonable choice.

This feature is leveraged as follows. Intel provides the `INVPCID` instruction that can be used to invalidate all the entries having a given PCID. This instruction needs to be used when the task finally terminates. Note that there is no need to flush the TLB or remove entries when there is a user process switch with the PCID mechanism.

Lazy TLB Mode

Let us now consider the case of multithreaded processes that run multiple threads across different cores. They share the same virtual address space, and it is important that if any TLB modification is made on one core, then the modification is sent to the rest of the cores to ensure program consistency and correctness. For instance, if a certain mapping is invalidated/removed, then it needs to be removed from the page table, and it also needs to be removed from the rest of the TLBs (on the rest of the cores). This requires us to send many inter-processor interrupts (IPIs) to the rest of the cores such that they can run the appropriate kernel handler and remove the TLB entry. As we would have realized by now, this is an expensive operation. It may interrupt a lot of high-priority tasks.

Consider a CPU that is currently executing another process. Given that it is not affected by the invalidation of the mapping, it need not invalidate it immediately. Instead, we can set the CPU state to the "lazy TLB mode".

> **Point 6.2.2**
>
> Note that kernel threads do not have separate page tables. A common kernel page table is appended to all user-level page tables. At a high level, there is a pointer to the kernel page table from every user-level page table. Recall that the kernel and user virtual addresses only differ in their highest bit (MSB bit), and thus a pointer to the kernel-level page table needs to be there at the highest level of the five-level composite page table.

Let us now do a case-by-case analysis. Assume that the kernel in the course of execution tries to access the invalidated page – this will create a correctness issue if the mapping is still there. Note that since we are in the lazy TLB mode, the mapping is still valid in the TLB of the CPU on which the kernel thread is executing. Hence, in theory, the kernel may access the user-level page that is not valid at the moment. This operation should not be allowed.

Note that access to user-level pages does not happen arbitrarily. Instead, such accesses happen via functions with well-defined entry points in the kernel. Some examples of such functions are `copy_from_user` and `copy_to_user`. At these points, special checks can be made to find out if the pages that the kernel

is trying to access are currently valid or not. If they are not valid because another core has invalidated them, then an exception needs to be thrown.

Next, assume that the kernel switches to another user process. In this case, either we flush all the pages of the previous user process (solves the problem) or if we are using ASIDs, then the pages remain but the current task's ASID/PCID changes. There is thus no correctness issue. Now consider shared memory-based inter-process communication that involves the invalidated page. This happens through well-defined entry points. Here checks can be carried out – the invalidated page will thus not be accessed.

Finally, assume that the kernel switches back to a thread that belongs to the same multithreaded user-level process. In this case, prior to doing so, the kernel checks if the CPU is in the lazy TLB mode and if any TLB invalidations have been *deferred*. If this is the case, then all such deferred invalidations are completed immediately prior to switching out from the kernel mode. This finishes the work.

The sum total of this discussion is that to maintain TLB consistency, we do not have to do it in mission mode. There is no need to immediately interrupt all the other threads running on the other CPUs and invalidate some of their TLB entries. Instead, this can be done lazily and opportunistically, as and when there is sufficient computational bandwidth available – critical high-priority processes need not be interrupted for this purpose.

6.2.5 Partitioning Physical Memory

NUMA Machines

Let us now look at partitioning physical memory. The kernel typically does not treat all the physical memory or the physical address space as a flat space, even though this may be the case in many simple embedded architectures. However, in a large server-class processor, this is often not the case, especially when we have multiple chips on the motherboard. In such a nonuniform memory access (NUMA) machine, where we have multiple chips and computing units on the motherboard, some memory chips are closer than the others to a given CPU. Clearly the main memory latency is not the same and there is a notion of memory that is *close* to the CPU versus memory that is far away in terms of the access latency. There is thus a non-uniformity in the main memory access latency, which is something that the OS needs to leverage for guaranteeing good performance.

Refer to Figure 6.11 that shows a NUMA machine where multiple chips (group of CPUs) are connected over a shared interconnect. They are typically organized into clusters of chips/CPUs and there is a notion of local memory within a cluster, which is much faster than remote memory (present in another cluster). We would thus like to keep all the data and code that is accessed within a cluster to remain within the local memory. We need to minimize the number of remote memory accesses as far as possible. This needs to be explicitly done to guarantee the locality of data and ensure a lower average memory access time. In the parlance of NUMA machines, each cluster of CPUs or chips is known as a *node*. All the computing units (e.g. cores) within a node have roughly the same access latency to local memory as well as remote memory. We need to thus organize the physical address space *hierarchically*.
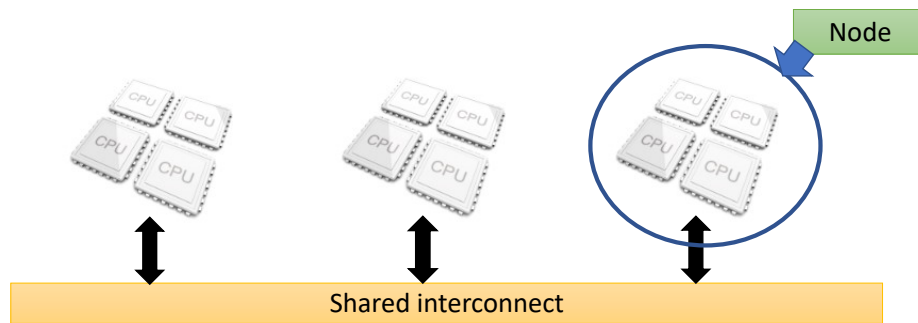
Figure 6.11: NUMA machine

Zones

Given that the physical address space is not *flat*, there is a need to partition it. Linux refers to each partition as a *zone* [Rapoport, 2019]. The aim is to partition the set of physical pages (frames) in the physical address space into different non-overlapping sets.

Each such set is referred to as a *zone*. They are treated separately and differently. This concept can easily be extended to also encompass frames that are stored on different kinds of memory devices. We need to understand that in modern systems, we may have memories of different types. For instance, we could have regular DRAM memory, flash/NVMe drives, plug-and-play USB memory, and so on. This is an extension of the NUMA concept where we have different kinds of physical memories, and they clearly have different characteristics with respect to the latency, throughput and power consumption. Hence, it makes a lot of sense to partition the frames across the devices and assign each group of frames (within a memory device) to a *zone*. Each zone can then be managed efficiently and appropriately (according to the device that it is associated with). Memory-mapped I/O and pages reserved for communicating with the DMA controller can also be brought within the ambit of such zones.

Listing 6.6 shows the details of the enumeration type `zone_type`. It lists the different types of zones that are normally supported in a regular kernel.

The first is `ZONE_DMA`, which is a memory area that is reserved for physical pages that are meant to be accessed by the DMA controller. It is a good idea to partition the memory and create an exclusive region for the DMA controller. It can then access all the pages within its zone freely, and we can ensure that data in this zone is not cached. Otherwise, we will have a complex sequence of cache evictions to maintain consistency with the DMA device. Hence, partitioning the set of physical frames helps us clearly mark a part of the memory that needs to remain uncached as is normally the case with DMA pages. This makes DMA operations fast and reduces the number of cache invalidations and writebacks substantially.

Next, we have `ZONE_NORMAL`, which is for regular kernel and user pages.

Sometimes we may have a peculiar situation where the size of the physical memory actually exceeds the total size of the virtual address space. This can happen on some older processors and also on some embedded systems that use 16-bit addressing. In such special cases, we would like to have a separate zone

of the physical memory that keeps all the pages that are currently not mapped to virtual addresses. This zone is known as `ZONE_HIGHMEM`.

User data pages, anonymous pages (stack and heap), regions of memory used by large applications, and regions created to handle large file-based applications can all benefit from placing their pages in contiguous zones of physical memory. For example, if we want to design a database's data structures, then it is a good idea to create a large folio of pages that are contiguous in physical memory. The database code can lay out its data structures accordingly. Contiguity in physical addresses ensures better prefetching performance. A hardware prefetcher can predict the next frame very accurately. The other benefit is a natural alignment with huge pages, which leads to reduced TLB miss rates and miss penalties. To create such large contiguous regions in physical memory, pages have to be freely movable – they cannot be pinned to physical addresses. If they are movable, then pages can dynamically be consolidated at runtime and large holes – contiguous regions of free pages – can be created. These holes can be used for subsequent allocations. It is possible for one process to play spoilsport by pinning a page. Most often these are kernel processes. These actions militate against the creation of large contiguous physical memory regions. Hence, it is a good idea to group all movable pages and assign them to a separate zone where no page can be pinned. Linux defines such a special zone called `ZONE_MOVABLE` that comprises pages that can be easily moved or reclaimed by the kernel.

The next zone pertains to novel memory devices that cannot be directly managed by conventional memory management mechanisms. This includes parts of the physical address space stored on nonvolatile memory devices (NVMs), memory on graphics cards, Intel's Optane memory (persistent memory) and other novel memory devices. A dedicated zone called `ZONE_DEVICE` is thus created to encompass all these physical pages that are stored on a device that is not conventional DRAM.

Such unconventional devices have many peculiar features. For example, they can be removed at any point of time without prior notice. This means that no copy of pages stored in this zone should be kept in regular DRAM – they will become inconsistent. Page caching is therefore not allowed. This zone also allows DMA controllers to directly access device memory. The CPU need not be involved in such DMA transfers. If a page is in `ZONE_DEVICE`, we can safely assume that the device that hosts the pages will manage them.

It plays an important role while managing nonvolatile memory (NVM) devices. All its constituent frames are mapped to this zone and there is a notion of isolation between device pages and regular memory pages. The key idea here is that device pages need to be treated differently in comparison to regular pages stored on DRAM because of device-specific idiosyncrasies.

Point 6.2.3

NVM devices are increasingly being used to enhance the capacity of the total available memory. We need to bear in mind that nonvolatile memory devices are in terms of performance between hard disks and regular DRAM memory. The latency of a hard disk is in milliseconds, whereas the latency of nonvolatile memory is typically in microseconds or in the 100s of nanoseconds range. The DRAM memory on the other hand has a sub 100-ns latency. The advantage of nonvolatile memories

is that even if the power is switched off, the contents still remain in the device (persistence). The other advantage is that it also doubles up as a storage device and there is no need to actually pay the penalty of page faults when a new process starts or the system boots up. Given the increasing use of nonvolatile memory in laptops, desktops and server-class processors, it was incumbent upon Linux developers to create a device-specific zone.

Listing 6.6: The list of zones
source : include/linux/mmzone.h#L610

```c
enum zone_type {
    /* Physical pages that are only accessible via the DMA
        controller */
    ZONE_DMA,

    /* Normal pages */
    ZONE_NORMAL,

    /* Useful in systems where the physical memory exceeds
        the size of max virtual memory.
        We can store the additional frames here */
#ifdef CONFIG_HIGHMEM
    ZONE_HIGHMEM,
#endif

    /* It is assumed that these pages are freely movable and
        reclaimable */
    ZONE_MOVABLE,

    /* These frames are stored in novel memory devices like
        NVM devices. */
#ifdef CONFIG_ZONE_DEVICE
    ZONE_DEVICE,
#endif

    /* Dummy value indicating the number of zones */
    __MAX_NR_ZONES
};
```

Sections

Recall that in Listing 6.5, we had talked about converting page frame numbers to `page` structures and vice versa. We had discussed the details of a simple linear layout of page structures and then a more complicated hierarchical layout that divides the zones into *sections*.

It is necessary to take a second look at this concept now (refer to Figure 6.12). To manage all the memory and that too efficiently, it is necessary to sometimes divide it into sections and create a 2-level hierarchical structure. The first reason is that we can efficiently manage the list of free frames within a section

because we use smaller data structures. Second, sometimes zones can be non-contiguous. It is thus a good idea to break a non-contiguous zone into a set of *sections*, where each section is a contiguous chunk of physical memory. Finally, sometimes there may be intra-zone heterogeneity in the sense that the latencies of different memory regions within a zone may be slightly different in terms of performance or some part of the zone may be considered to be volatile, especially if the device tends to be frequently removed.

Given such intra-zone heterogeneity, it is a good idea to partition a zone into sections such that different sections can be treated differently by the kernel and respective memory management routines. Next, recall that the code in Listing 6.5 showed that each section has its `mem_map` that stores the mapping between page frame numbers (pfns) and struct pages. This map is used to convert a pfn to a struct page.
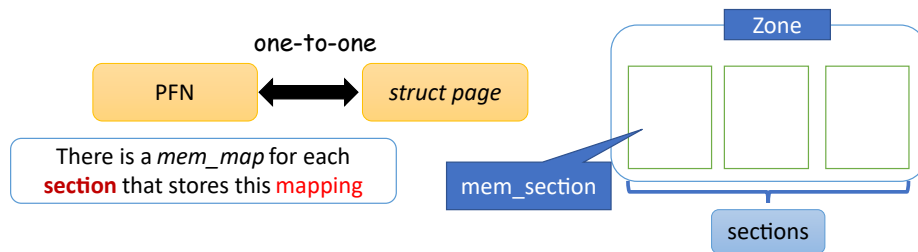
Figure 6.12: Zones and sections

Detailed Structure of a Zone

Now, let us look at the structure of a zone (`struct zone`) shown in Listing 6.7. Each zone is associated with a NUMA node (`node` field), whose details are stored in the `pglist_data` structure.

Each zone has a starting page frame number and an ending page frame number. The starting page frame number is stored in `zone_start_pfn`. The field `spanned_pages` is used to compute the last page frame number of the zone. It is important to note that it does not represent the size of the zone. The number of pages in the zone is instead stored in a separate field `present_pages`. In the case of `spanned_pages`, we use it as follows. The ending page frame number is `zone_start_pfn + spanned_pages - 1`. If the zone is contiguous then `present_pages` is equal to `spanned_pages`, otherwise they are different.

The field `managed_pages` refers to the pages that are *actively managed* by the kernel. This is needed because there may be a lot of pages that are a part of the zone, but the kernel is currently not taking any cognizance of them and in a sense is not managing them.

Next, we store the name of the zone and also have a hierarchical list of free regions within a zone (`free_area[]`). `free_area[]` is used by the buddy allocator (see Section 6.4.1) to allocate contiguous memory regions in the physical address space.

Listing 6.7: `struct zone`
source : `include/linux/mmzone.h#L705`

```
struct zone {
    int node;  /* NUMA node */

    /* Details of the NUMA node */
    struct pglist_data  *zone_pgdat;

    /* zone_end_pfn = zone_start_pfn + spanned_pages - 1 */
    unsigned long      zone_start_pfn;
    atomic_long_t      managed_pages;
    unsigned long      spanned_pages;
    unsigned long      present_pages;

    /* Name of the zone */
    const char         *name;

    /* List of the free areas in the zone (managed by the
       buddy allocator) */
    struct free_area   free_area[MAX_ORDER];
}
```

Details of a NUMA Node

Let us now look at the details of a NUMA node (see Listing 6.8). `struct pglist_data` stores the relevant details. A NUMA node is identified by its node ID (`node_id`). Each node can contain several zones. These are stored in the array `node_zones`. We can have one zone of each type at the most. The number of populated zones in a node is given by the `nr_zones` variable.

On the other hand, `node_zonelists` contains references to zones in all the nodes. This structure contains global information across the system. In general, the convention is that the first zone in each zone list belongs to the current node. The present and spanned pages retain the same meanings.

We would like to specifically point out two more important fields. The first is a pointer to a special kernel process `kswapd`. It is a background process, also known as a daemon, that finds infrequently used pages and migrates them to the swap space. This frees up much-needed memory space. Additionally, on a NUMA machine, it migrates pages across NUMA nodes based on their access patterns. This is a rather low-priority process that runs in the background but nevertheless does a very important job. It frees up memory and balances the used memory across NUMA nodes.

The field `__lruvec` refers to LRU-related information that is very useful for finding the pages that have been infrequently used in the recent past. This is discussed in great detail in Section 6.3.2.

<div align="center">

Listing 6.8: `struct pglist_data`

source : include/linux/mmzone.h#L1121

</div>

```
typedef struct pglist_data {
    /* NUMA node id */
    int node_id;

    /* Hierarchical organization of zones */
    struct zone node_zones[MAX_NR_ZONES];
```

```
    struct zonelist node_zonelists[MAX_ZONELISTS];
    int nr_zones;

    /* #pages owned by the NUMA node (node_id) */
    unsigned long node_present_pages;
    unsigned long node_spanned_pages;

    /* Pointer to the page swapping daemon */
    struct task_struct    *kswapd;

    /* LRU state information */
    struct lruvec         __lruvec;
} pg_data_t;
```

6.3   Page Management

Let us start with describing the necessary background for understanding the page management data structures in the kernel. First, we need to understand Bloom filters described in Section C.6.1 of Appendix C. The other basic concept is reverse mapping, which we shall cover in this section. Then we shall move on to describe the MGLRU (multi-generation LRU) page replacement algorithm that Linux uses.

Note that this section will use Bloom filters heavily and thus reading it from the appendix is essential. It is a probabilistic data structure that is used to test for set membership. It answers the basic query, "Is x an element of set $\mathcal{S}$?" It allows false positives but never allows false negatives. Gradually, as more items are added to the Bloom filter, the probability of false positives increases, hence after a point there is a need to reset it – clear all its entries.

6.3.1   Reverse Mapping

The most important piece of background that we need to introduce is the concept of em reverse mapping. Here, we map physical pages to virtual pages. It could be a one-to-many mapping. In a virtual memory system, it is necessary to have such a reverse mapping because if a physical page is moved or swapped out, then the corresponding page tables of all the processes that have an entry pointing to it need to be updated. For example, if a physical page P is mapped in two processes A and B, and P is swapped out, then there is a need to use this mechanism. We need to somehow remember the fact that two processes – A and B – use the physical page P. Once, P is swapped out or its protection bits are changed, we need to first find all the processes that have a mapping for it. In this case, they would be A and B. Next, we need to access the page tables of A and B and make the appropriate changes.

In this context, note that there is a fair amount of physical page sharing in Linux even when shared memory is not used. This is because processes are created by making *fork* calls. Each such call begins with sharing all the pages in COW (copy-on-write) mode. Unless a later *exec* call loads a fresh set of pages, a fair amount of residual page sharing still remains.

Before we start, let us refresh our memory. Recall the core concept of a *vma region* (struct vm_area_struct) introduced in Section 3.1.10. It points to a

contiguous region of virtual memory. It stores the starting and ending virtual addresses, type of the region, etc. Page sharing between processes happens at the granularity of these `vma` regions. We shall see that a `vma` region makes it easier to manage sharing. Specifically, we can have two kinds of `vma` regions or pages: anonymous pages and file-backed pages. Anonymous pages correspond to data that is generated in the course of execution. Stack and heap pages are anonymous pages. File-backed pages correspond to memory-mapped files. They are typically stored and managed by a page cache.

Let us discuss the reverse mapping problem for anonymous pages. For file-backed pages, solving this problem is easier to solve. Here, we use a centralized structure that maintains a tree of `vmas` that are mapped to bytes in the file. Given a physical address, we can map it to a fixed file offset. Then, we can use that information to find the `vmas` that may contain the physical address. The general problem for anonymous pages is much harder.

Background of the Problem

Given a physical page, we need to find the processes that have mappings for it. Given the physical address, we have already seen how to map it to a `struct page` in Section 6.2.3. A straightforward solution appears to be that we store a list of processes (`task_structs`) within each `struct page`. This is a bad idea because there is no limit on the number of processes that can map to a page. We thus need to store the list of processes in a linked list. However, it is possible that a lot of space is wasted because numerous pages may be shared in an identical manner. We will end up creating many copies of the same linked list.

Moreover, many a time there is a requirement to apply a common policy to a group of pages. For example, we may want all the pages to have a given access permission. Given that we are proposing to treat each page independently, this is bound to be quite difficult. We thus have two problems at hand.

Mapping Problem For every page, we need to have a linked list of virtual pages.

Policy Problem It should be easy to set a common policy for a group of pages.

Let us propose a solution to the Policy Problem. We will then introduce a few tricks and the Mapping Problem will get solved on its own !!!

Let us not have a linked list associated with each `struct page`. This is not a scalable solution. This will create a need to go and change the attributes of every virtual page one by one, in case we want to apply a certain access policy to a large memory region. Let us instead focus on the Policy Problem. We can add a pointer to a `vma` (vm area structure) in each `struct page`. Accessing the `vma` that contains a page is thus quite easy. The common policy and access permissions can be stored in the associated `vma`.

This may appear to be a good solution on cursory examination. However, it is quite problematic. A physical page may map to multiple `vmas` across processes. Hence, having just a single pointer does not help. Also, `vmas` tend to get split and merged quite frequently. This can happen due to changing the access permissions of memory regions (due to allocation or deallocation), moving groups of pages between different regions of a program or enforcing

different policies for different regions within a `vma`. Examples of the last category include different policies with regard to prefetching, swapping priorities and making pages part of huge pages. `vmas` can also get copied in the case of *fork* operations. This will create a one-to-many mapping between physical pages and `vmas`. Given the fluid nature of `vmas`, it is not advisable to directly add a pointer to a `vma` in each `struct page`. If there is a change, then we need to walk through the `page` structures corresponding to all the constituent pages of a `vma` and make changes. This will take $O(N)$ time. We desire a solution that takes O(1) time.

Notion of the anon_vma

In all such cases, we leverage a standard pattern namely *virtualization*. The idea is to *virtualize* a `vma` by adding another data structure in the middle. This is a standard technique that we have used in many other places as well. The designers of Linux introduced the `anon_vma` structure that introduces an additional level of indirection. Each physical page now points to an `anon_vma`, which is nothing but a dummy stub. The `anon_vma` is accessible via the `mapping` field (see Figure 6.13) in a `struct page`. Now a group of physical pages can point to the same `anon_vma`. The `anon_vma`, in turn, can point to a set of `vmas`. The idea here is that a group of pages are shared across a set of processes – each with its own `vma`. Policies can be enforced by each process independently. It can record the policy in its private `vma`.
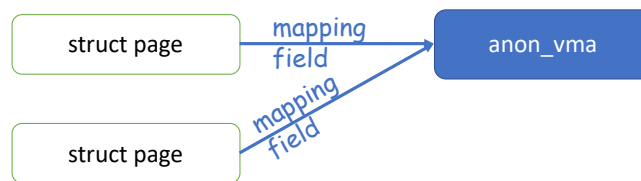
Figure 6.13: Pages pointing to an `anon_vma`

Each `anon_vma` is thus associated with a set of `vmas`. This is easily understood given that when a process is forked, an additional `vma` is created in the child process for each `vma` in the parent process. They are identical. Now both the `vmas` – one that corresponds to the parent and one that corresponds to the child – are associated with the same set of pages. This basically means that an `anon_vma` that represents a subset of these pages is now associated with two `vmas` across two processes (the parent and the child). The relationship is as follows: $\boxed{2~\texttt{vma} \leftrightarrow 1~\texttt{anon\_vma}}$ (refer to Figure 6.14(a)). The advantage of adding a new level of indirection in terms of the `anon_vma` is that no page needs to change its mapping. We only need to tweak the mapping between an `anon_vma` and `vmas` of processes. New processes may start via forking and old processes may terminate. All of these changes can be easily taken care of at this level. Given that the number of `vmas` is far lower than the number of pages, this is going to be very fast.

Now, consider another case, which is more complicated. Consider a case where a parent process has forked a child process. In this case, they have their separate `vmas` that point to the same `anon_vma`. This is the one that the shared

pages also point to. Now, assume a situation where the child process writes to a page that is shared with the parent. This means that a new copy of the page has to be created for the child due to the copy-on-write (CoW) mechanism. This new page needs to point to an `anon_vma`, which clearly cannot be the one that the previously shared page was pointing to. This is because the new page is in the private address space of the child process. Hence, it needs to point to a new `anon_vma` that corresponds to pages exclusive to the child. There is an important question that needs to be answered here. What happens to the child's `vma`? Assume it had 1024 pages, and the write access was made to the 500th page. Do we then split it into three parts: 0-499, 500, 501-1023? The first and last chunks of pages are unmodified up till now. However, we made a modification in the middle, i.e., to the 500th page. This page is now pointing to a different `anon_vma`.
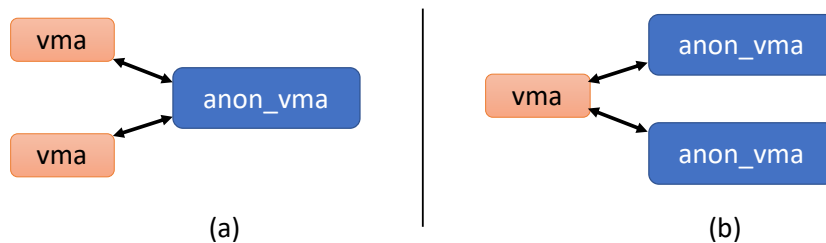
Figure 6.14: (a) Many-to-one mapping (b) One-to-many mapping

Splitting the `vma` is not a good idea. This is because a lot of pages in a `vma` may see write accesses when they are in a copy-on-write (COW) mode. We cannot keep on splitting the `vma` into smaller and smaller chunks. This is a lot of work and will prohibitively increase the number of `vma` structures that need to be maintained. Hence, as usual, the best solution is to do nothing, i.e., not split the `vma`. Instead, we maintain the `vma` of the child process, as it is, but assume that all the pages in its range may not be mapped to the same `anon_vma`. Some may point to the parent process's `anon_vma` and some may point to the child process's `anon_vma`. We thus have the following relationship: $\boxed{1 \text{ vma} \leftrightarrow 2 \text{ anon\_vma}}$. Recall that we had earlier shown a case where we had the following relationship: $\boxed{2 \text{ vma} \leftrightarrow 1 \text{ anon\_vma}}$ (refer to Figure 6.14(b)).

Let us now summarize our learnings.

> **Point 6.3.1**
>
> This is what we have understood about the relationship between a `vma` and `anon_vma`.
>
> - For a given virtual address region, every process has its own private `vma`. It is stored in the maple tree of `mm_struct`.
>
> - A physical page points to only a single `anon_vma`. The `anon_vma` acts as a stub.
>
> - Multiple `vma` structures across processes need to point to a single `anon_vma` owing to the fork operation.

- A copy-on-write operation causes a page to change its `anon_vma`. However, the `vma` structures should be kept intact in the interest of time. The page now points to an `anon_vma` corresponding to the process's private pages.

- `vma` regions can be dynamically split or merged depending on user actions such as changing the policies or access permissions with respect to a set of memory addresses.

- Summary: There is a many-to-many relationship between `vma` and `anon_vma` structures.
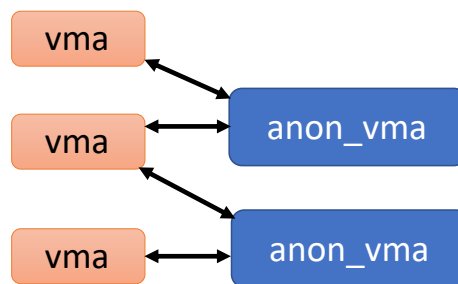
Figure 6.15: Example of a many-to-many mapping

We thus observe that a complex relationship between the `anon_vma` and `vma` has developed by this point (refer to Figure 6.15 for an example). Maintaining this information and minimizing runtime updates to such structures is not easy. There is a classical time and space trade-off here. If we want to minimize time, we should increase space. Moreover, we desire a data structure that captures the dynamic nature of the situation. The events of interest that we have identified up till now are as follows: a fork operation, a write to a COW page, splitting or merging a `vma` and killing a process.

Let us outline our requirements using a few one-line principles.

1. Every `anon_vma` should know which `vma` structures it is associated with.

2. Every `vma` should know which `anon_vma` structures it is associated with.

3. The question that we need to answer now is whether these two structures should directly point to each other or is another intermediate structure required? In other words, is another additional level of indirection between `vma` and `anon_vma` required?

We shall show the C code of an `anon_vma` after we describe another structure known as the `anon_vma_chain`, which is an intermediate structure.

anon_vma_chain

The question that we need to answer here is whether we should have an array of `vma` pointers in each `anon_vma` and an array of `anon_vma` pointers in each `vma`

to implement a many-to-many mapping? This seems to be the most straightforward solution. The kernel developers did think about this, but they found scalability problems [Corbet, 2010].

Consider a linked list of `vma` structures. This linked list can be reasonably large if there is a lot of sharing, especially in enterprise-scale applications. To make updates to a `vma`, there is a need to lock sections of the linked list along with the `vma` structure. If there are concurrent updates to `vma` structures, then scalability is limited. It is not possible to make many concurrent modifications.

This is where another level of indirection between the `vma` and `anon_vma` will help. The Linux developers thus proposed a simple data structure that embodies the connection between a `vma` and `anon_vma`. It is called an `anon_vma_chain`.

It is designed as shown in Listing 6.9 (pictorially represented in Figure 6.16).

Listing 6.9: `anon_vma_chain`
source : include/linux/rmap.h#L82

```c
struct anon_vma_chain {
    struct vm_area_struct *vma;    /* pointer to a vma */
    struct anon_vma *anon_vma;     /* pointer to an anon_vma
        */

    struct list_head same_vma;     /* pointer to other
        anon_vma_chains corresponding to the same task */
    struct rb_node rb;             /* pointer to a node in
        the red-black tree */
};
```


Figure 6.16: The relationship between `vma`, `anon_vma` and `anon_vma_chain` (`avc`). A dashed arrow indicates that `anon_vma` does not hold a direct pointer to the `avc`, but holds a reference to a red-black tree that in turn has a pointer to the `avc`. This is a secondary detail. For all practical purposes, an `anon_vma` has a pointer to each `avc` that points to it.

We can think of the `anon_vma_chain` (`avc`) structure as a link between a `vma` and an `anon_vma`. We had aptly referred to it as a level of *indirection*. An advantage of this structure is that we can create a logical separation between a `vma` and `anon_vma`. Processes can work on the `vma` without worrying about its connections with `anon_vmas`. Adding connections is quite easy now. We just need to add a new `anon_vma_chain` node to a linked list that is maintained by each `vma`. These `avc` nodes store pointers to `anon_vmas` that are possibly associated with pages that map to the address encompassed by the `vma`.

Let us now get into the specifics. Each `vma` has an `avc` as its member. It is a part of a linked list of `avcs`. Each `avc` has a member called `same_vma` that links it to other nodes of a doubly linked list of `avcs` (refer to Figure 6.17). All of these `avcs` correspond to the same `vma`.

> **Point 6.3.2**
>
> A virtual memory region in a task is represented by the `vma` structure. The physical pages that map to this virtual memory region may have been allocated by different processes such as the current process and ancestor processes. They thus point to different `anon_vmas`. However, there is a need to link all of them together and associate them with the current `vma`. This is achieved effectively by creating a link structure called an `anon_vma_chain` (abbreviated as `avc`).
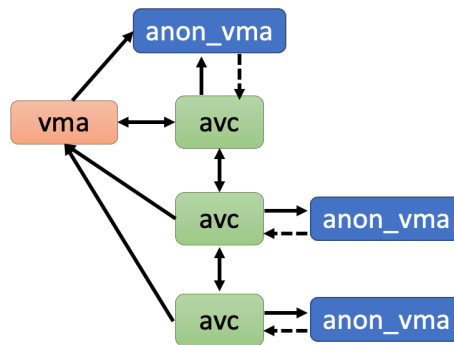
Figure 6.17: `anon_vma_chain` nodes connected together as a list (all corresponding to the same `vma`)

anon_vma

Now, let us look at the code for the `anon_vma` in Listing 6.10.

Listing 6.10: `anon_vma`
source : include/linux/rmap.h#L31

```
struct anon_vma {
    struct anon_vma *root;
    struct anon_vma *parent;

    struct rb_root_cached rb_root;    /* interval tree of
        anon_vmas */
}
```

We don't actually store any state. It is indeed a dummy node. We just store pointers to other data structures. All that we want is that all the pages in a virtual memory region that were allocated by the same process point to the same `anon_vma`. Now, given an `anon_vma`, we need to quickly access all the `vma` structures that may contain a mapping to any page that points to it (across processes). Then we need to solve the reverse mapping problem. This has two steps. ① We first locate the `vma` that may contain a mapping to the physical page. ② Next, we find the virtual page within the `vma` that is mapped to the given physical page. Let us solve problem ① first.

Before we proceed to understand the code, we need to appreciate that Linux relies a lot on storing data implicitly. Recall the design of the linked list. A structure just stores a generic linked list node. It then traverses the linked list and use a macro to reach the first byte of the encapsulating structure of a linked list node. Hence, we need to always proceed bearing in mind that all the information may not be visible to us.

The problem is as follows. Given a `struct page`, we find its associated `anon_vma`. Each physical page points to a <u>single</u> `anon_vma`. From the `anon_vma`, we need to reach all the `vmas` (across processes) that possibly map the page. To facilitate this, every `anon_vma` points to the root of a red-black tree via its member `rb_root`. Each node of this tree is an `anon_vma_chain` node (`anon_vma_chain− > rb`). The red-black tree can quickly be used to find all the `vma` structures that contain a page. Additionally, we organize all the `anon_vma` structures as a tree, where each node points to a parent `anon_vma` and the root of the tree.

Let us explain with an example. Consider a process where all its pages are mapped to its `anon_vma` node. These pages were private to it. Now it got forked. Then in the child process, a new `avc` will be created to point to the `anon_vma` of the parent. This means that at this point of time, the red-black tree corresponding to the parent has two `avcs`: one belongs to the parent and one belongs to the child. Furthermore, the child process will have its `anon_vma`. Given that it arose out of the parent, a parent pointer is added to the `anon_vma` of the parent.

We shall look at more extensive examples later.

Revisiting the `vma`

Let us revisit the `vm_area_struct` (`vma`) structure that stores information regarding a virtual memory region. We have been referring to its short form `vma` all along. It needs to now be linked with all the additional data structures that we have just created.

As we have discussed, a `vma` is not directly pointed to by pages. The pages instead point to an `anon_vma` that is associated with a `vma`. This means that each `vma` needs to have at least one dedicated `anon_vma` associated with it and an `anon_vma_chain` node that connects them both. Figure 6.18 reflects this relationship.

For storing both these fields of information, the `vma` has two fields (refer to Listing 6.11).

<div align="center">

Listing 6.11: `anon_vma_chain`

source : include/linux/mm_types.h#L567

</div>

```
struct vm_area_struct {
    ...
    struct list_head anon_vma_chain;
    struct anon_vma *anon_vma;
    ...
}
```

The private `anon_vma` is named `anon_vma` and the list of `anon_vma_chain` nodes is represented by its namesake `anon_vma_chain.`

As we have discussed, it is possible that because of fork operations, many other `vma` structures (across child processes) are associated with the same `anon_vma`
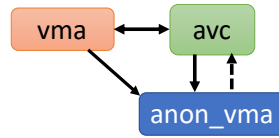
Figure 6.18: Updated relationship between `vma`, `anon_vma` and `avc`

via `avcs`. This is the `anon_vma` that is associated with the `vma` of the parent process. Figure 6.19 shows an example where the parent process's `anon_vma` is associated with multiple `vmas` across child processes via `avcs`.
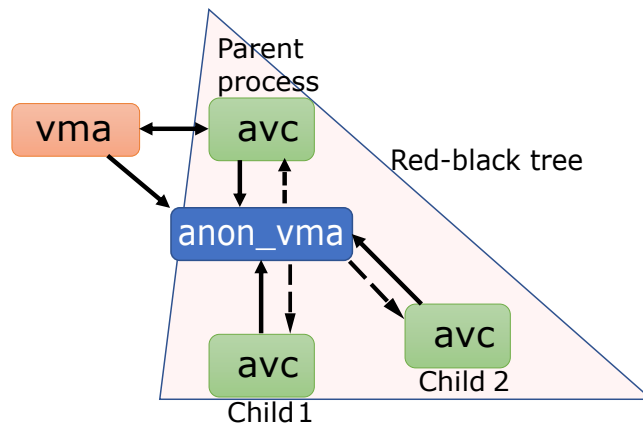
Figure 6.19: Example of a scenario with multiple processes where the parent `anon_vma` is associated with multiple child `vmas`.

Explanation with Examples: *fork* and *copy-on-write*
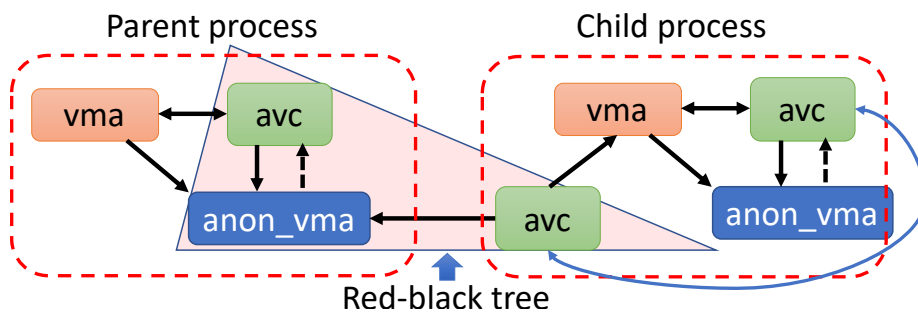
Figure 6.20: Reverse map structures after a *fork* operation

Let us now consider the case of a *fork* operation. The reverse map (rmap) structures are shown in Figure 6.20 for both the parent and child processes. The parent process has one `vma` and an associated `anon_vma`. The *fork* operation

starts out by creating a copy of all the *rmap* structures of the parent. The child thus gets an `avc` that links its `vma` to the parent's `anon_vma`. This ensures that all shared pages point to the same `anon_vma` and the structure is accessible from both the child and parent processes.

The child process also gets its own private `anon_vma` that is pointed to by its `vma`. This is for pages that are exclusive to it and not shared with its parent process.

Given a page frame number, we can locate its associated `struct page`, and then using its `mapping` field, we can retrieve a pointer to the `anon_vma`. The next problem is to locate all the `avcs` that point to the `anon_vma`. As discussed before, every `anon_vma` points to a red-black tree that stores the set of `avcs` that point to it.

The child process has two `avcs`. They are connected to each other and the child process's `vma` using a doubly linked list.
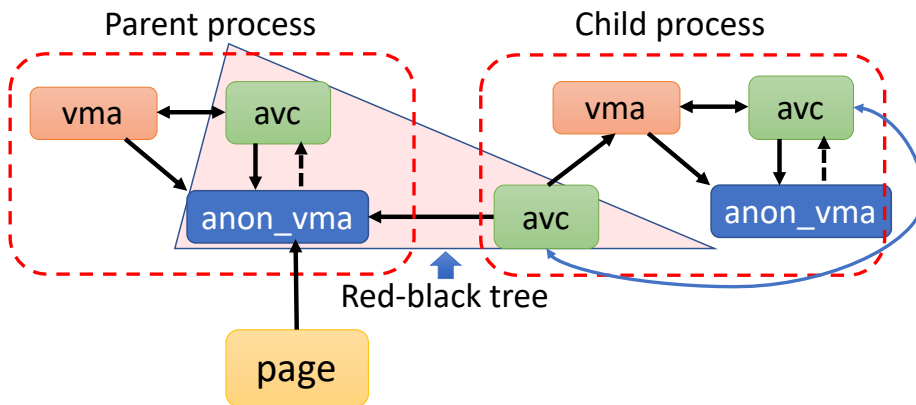
Figure 6.21: A page pointing to the parent's `anon_vma`

Let us now consider the case of a shared page that points to the `anon_vma` of the parent process (refer to Figure 6.21). After a *fork* operation, this page is stored in copy-on-write (COW) mode. Assume that the child process writes to the page. In this case, a new copy of the page needs to be made and attached to the child process. This is shown in Figure 6.22.

The new page now points to the private `anon_vma` of the child process. It is now the exclusive property of the child process.

Next, assume that the child process is forked. In this case, the rmap structures are replicated, and the new grandchild process is also given its private `vma` and `anon_vma` (refer to Figure 6.23). In this case, we create two new `avcs` for the grandchild. One `avc` points to the `anon_vma` of the child process and the other `avc` points to the `anon_vma` of the original parent process. The figure shows two red-black trees: one corresponds to the parent process's `anon_vma` and the other corresponds to the child process's `anon_vma`. The `avcs` of each process are also nicely linked using a doubly linked list, which also includes its `vma`.

After repeated *fork* operations, it is possible that a lot of `avcs` and `anon_vmas` get created. This can lead to a storage space blowup. Modern kernels optimize this. Consider the `anon_vma` (+ associated `avc`) that is created for a child process
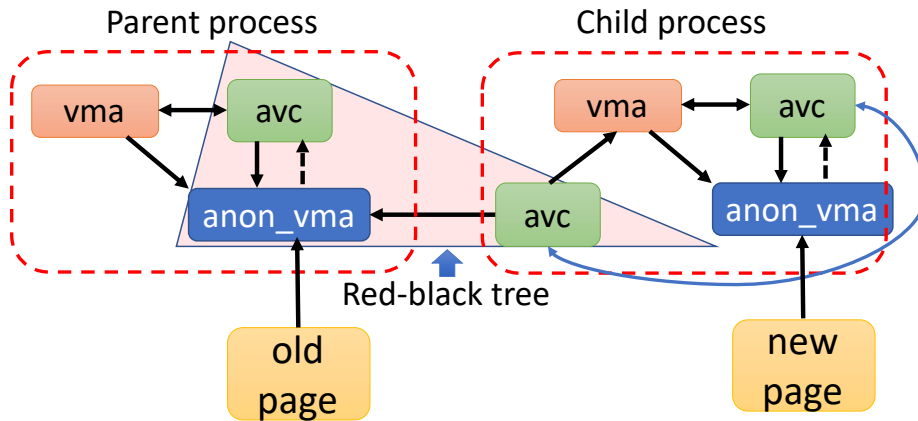
Figure 6.22: A new page pointing to the child's `anon_vma`
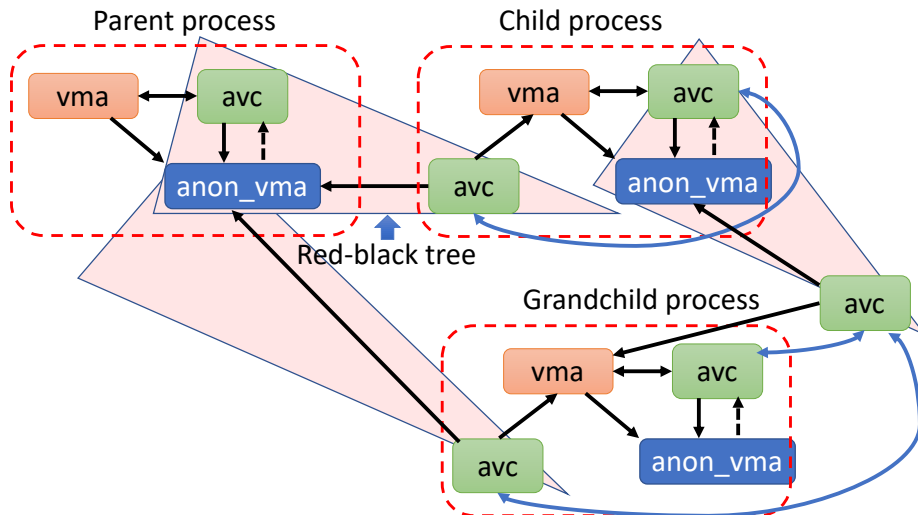
Figure 6.23: The structure of the rmap structures after a second *fork* operation (fork of the child)

such that pages that are exclusive to the child can point to it. In some cases, instead of doing this, an existing `anon_vma` along with its `avc` can be reused. This introduces an additional level of complexity; however, the space savings justify this design decision to some extent. This is quite complex. Hence, this is out of the scope of the book.

Notion of the Index

We have not solved the Mapping Problem completely. After locating a `vma`, we need to find the virtual page that maps to the physical page. We cannot afford to do a linear search, which would entail checking the page table of every virtual page in the `vma`. There needs to be a solution that runs in constant time.

Consider the first time that a physical page is allocated and mapped to a virtual page. Let the virtual page be a part of `vma` v. Assume that v contains N pages, and the ith page in v is mapped. We term i as the *index* of the physical page. This is stored in the associated `struct page`.

Next, if there is a need for reverse mapping, the physical page frame number p will be presented to `vma` v. Its associated `struct page` is retrieved and index i is used to compute the virtual address *vaddr*. The formula for computing *vaddr* is as follows.

$$vaddr = v.start\_addr + i << 12 \qquad (6.1)$$

Every `vma` has a starting virtual address. Subsequently, the `vma` corresponds to a range of contiguous virtual addresses. The ith page is an offset from the starting address of the `vma`. The starting address is $v.start\_addr$. We need to multiply i with the page size, 4 KB. This is the same as left-shifting i by 12 positions. Once, we have computed the virtual address, we can access its page table entry. If its page frame number is p, we have a match.

We can use the same logic while accessing `vmas` of child processes. They would be created by forking the parent. Hence, the child `vma` will have the same size as the parent `vma`. Even if the starting virtual address of the child `vma` is different, it does not matter. Equation 6.1 can still be used. Sometimes a `vma` can grow or shrink. This can happen to the `vma` of the parent process or the `vma` of the child process. Assume that a `vma` started out with containing 100 pages. Let us number them $0 \ldots 99$. It later gets shrunk and contains only the virtual pages 50-70. In this case, the numbers 50 and 70 are stored in the `vma` data structure. When it is presented with index i, we need to first check if it is between 50 and 70. If it is not between them, we can conclude that the `vma` does not contain any mapping for the page. Otherwise, we compute the virtual address using the following equation.

$$vaddr = v.start\_addr + (i - 50) << 12 \qquad (6.2)$$

If bookkeeping is done correctly, then the virtual address can easily be found out from the index in $\theta(1)$ time.

6.3.2 The MGLRU Algorithm for Page Replacement

Key Design Decisions

Let us first understand the key decisions that are made while designing a page replacement algorithm. The first objective is to leverage temporal and spatial locality, wherever it exists. Moreover, we need to ensure that the algorithm for finding pages to evict executes as quickly as possible. In this regard, a typical approach that is employed in most modern systems is the slow-path-fast-path method. We can create a fast method to satisfy most of the requests. In a small minority of the cases, there would be a need to use the much *slower path*.

The aim is to always create a design using simple heuristics that generalizes well. This is because, a general-purpose operating system needs to run many kinds of workloads, those that exist today and the ones that will be created in the foreseeable future.

The MG-LRU algorithm is a sophisticated variant of the WS-Clock Second Chance Algorithm that we studied in Section 6.1.2. "MG" stands for "multi-generation". Its key features are as follows:

- The algorithm divides pages into different *generations* based on the recency of the last access. If a page is accessed, there is a fast algorithm to upgrade it to the *latest generation*.

- The algorithm reclaims pages in the background by swapping them out to the disk. It swaps pages that belong to the oldest generations.

- It *ages* the pages very intelligently.

- It is tailored to running large workloads and integrates well with the notion of folios. Recall that a folio is a compound page. It contains multiple pages that are contiguous in both the virtual and physical address spaces.

Key Data Structures

struct lruvec

Listing 6.12: `struct lruvec`
source : include/linux/mmzone.h#L508

```
struct lruvec {
    /* contains the physical memory layout of the NUMA
        node */
    struct pglist_data *pgdat;

    /* Number of refaults */
    unsigned long refaults [ANON_AND_FILE];

    /* LRU state */
    struct lru_gen_struct    lrugen;
    struct lru_gen_mm_state  mm_state;
};
```

Linux uses the `lruvec` structure to store metadata. Its code is shown in Listing 6.12. The first field is a pointer to a `pglist_data` structure that stores the details of the zones in the current NUMA node (discussed in Sections 6.2.5 and 6.2.5).

Next, we store the number of *refaults* for anonymous and file-backed pages. A *refault* is a page access after it has been evicted. We clearly need to minimize the number of refaults. If it is high, it means that the page replacement and eviction algorithms are suboptimal – they evict pages that have a high probability of being accessed in the near future.

The next two fields `lrugen` and `mm_state` store important LRU-related state. `lrugen` is of type `lru_gen_struct` (shown in Listing 6.13). `mm_state` is of type `lru_gen_mm_state` (shown in Listing 6.14).

struct lru_gen_struct

Listing 6.13: `struct lru_gen_struct`
source : include/linux/mmzone.h#L407

```
struct lru_gen_struct {
    /* sequence numbers */
    unsigned long  max_seq;
    unsigned long  min_seq[ANON_AND_FILE];

    /* When was a generation created */
    unsigned long timestamps[MAX_NR_GENS];

    /* 3D array of lists */
    struct list_head  lists[MAX_NR_GENS][ANON_AND_FILE][
        MAX_NR_ZONES];
};
```

The crux of the MGLRU algorithm is the concept of a *generation*. Every generation is a sequence number. Higher the sequence number, more recent is the generation. `max_seq` is the largest sequence number. It corresponds to the latest generation. There are two types of minimum sequence numbers. We maintain minimum sequence numbers for file-backed pages and anonymous pages, respectively. Hence, we have a 2-element array `min_seq`.

The moment a page is accessed, it is promoted to a higher generation. It is obvious that we need to evict pages from the earliest generation (one with the minimum sequence number). Once all the pages in the earliest generation have been evicted, the minimum sequence number can be incremented. We would not like to evict pages in a newly-created generation. This is why there is a need to store the time at which a generation was created. This is stored in the `timestamps` array. No page should be evicted if its generation was created recently (less than a certain threshold time).

`lists` is a 3D array of linked lists. It is indexed by the generation number, type of page (anonymous or file) and zone number. Each entry in this 3D array is a linked list whose elements are `struct pages`. The idea is to link all the pages of the same type that belong to the same generation in a single linked list. We can traverse these lists to find pages to evict.

struct lru_gen_mm_state

Listing 6.14: `struct lru_gen_mm_state`
source : include/linux/mmzone.h#L444

```
    struct lru_gen_mm_state {
        unsigned long seq;

        /* Number of page walkers */
        int nr_walkers;

        /* head and tail pointers in the linked list */
        struct list_head *head;
        struct list_head *tail;

        /* array of Bloom filters */
        unsigned long *filters[NR_BLOOM_FILTERS];
    };
```

Let us next discuss the code of `lru_gen_mm_state` (see Listing 6.14). This structure stores the current state of a page walk – a traversal of all the pages to find pages that should be evicted and written to secondary storage (swapped out). At one point, multiple threads may be performing page walks (stored in the `nr_walkers` variable).

The field `seq` is the current sequence number that is being considered in the page walk process. Recall that each sequence number corresponds to a generation. The `head` and `tail` pointers point to consecutive elements of a linked list of `mm_struct` structures. In a typical page walk, we traverse all the pages that satisfy certain criteria of a given process, then we move to the next process (its `mm_struct`), and so on. This process is easily realized by storing a linked list of `mm_struct` structures. The `tail` pointer points to the `mm_struct` structure that was just processed (pages traversed). The `head` pointer points to the `mm_struct` that needs to be processed.

Finally, we use an array of Bloom filters to speed up the page walk process (we shall see later). Whenever, the word Bloom filter comes up, the only thing that one should have in mind is that in a Bloom filter a false negative is not possible, but a false positive is possible.

Page Access Tracking

Now that we have introduced the data structures used in the MGLRU algorithm, let us look at the missing piece. Prior to using the data structures, we need to have some basic information stored in each page table entry. We need to know if it has been recently accessed or not.

Listing 6.15: Tracking page accesses
source : arch/x86/include/asm/pgtable.h#L330

```
pte_t pte_mkold(pte_t pte)
{
    return pte_clear_flags(pte, _PAGE_ACCESSED);
}
```

A standard approach is to have a "recently accessed bit" in each page table entry. Periodically, all such bits are set to zero. Next, whenever a page is accessed, the corresponding bit is set to 1. Ideally, if hardware support is available, this can be done automatically. This would entail making the entire page table accessible to hardware. As of 2024, some x86 processors have this facility. They can automatically set this flag. A simple scan of the page table can yield the list of pages that have been recently accessed (after the last time that the bits were cleared). If hardware support is not available, then there is a need to mark the pages as *inaccessible*. This will lead to a soft page fault, when the page is subsequently accessed. This is not a full-scale (hard) page fault, where the contents of the page need to be read from an external storage device. It is instead, a soft page fault, where after recording the fact that there was a page fault, its access permissions are changed – the page is made accessible once again. We basically deliberately induce fake page faults to record page accesses. Recall that we had done something similar in Section refsec:softpagefault, when we had created a practical implementation of the LRU algorithm.

Regardless of the method used, once a page is accessed, the corresponding bit is set either automatically by hardware or by the fake page fault mechanism. Periodically, this bit needs to be cleared. This basically means that a page needs to be marked as "not accessed" or unused (refer to Listing 6.15). Such unused pages can subsequently be marked for eviction. We shall subsequently learn that we periodically or on-demand scan all the pages, mark them unused if required, and then ultimately evict them in due course of time.

Shrinking the Memory Footprint

Let us now discuss the situations when the LRU structures will be used to evict pages or reclaim (replace) pages.

The kernel maintains a dynamic count of the number of pages that are in main memory for each zone. It has an idea of the *page pressure* at all points of time. Whenever, there are too many pages in memory, there is a need to evict pages from memory and create space. There are two methods of dynamically shrinking the memory footprint of zones in main memory. There is a passive process and an active process.

The passive process uses a kernel daemon *kswapd*. Its job is to run periodically, *age* pages and evict them. The basic idea is to compact the memory footprint of different zones in main memory. All the zones are typically shrunk by different degrees. It makes a call to the function *evict_folios* that performs the role of eviction. This function can also be called directly by other kernel subsystems, especially if they feel that there are too many pages in memory. This comprises the *active process* of evicting pages.

At this point of time, it is important to introduce the term *page reclamation*. It is not the same as eviction – it is a more general term. Let us consider the regular case first. A physical page in main memory is used to store data and is mapped to a virtual page. In this case, reclamation simply means eviction. Let us consider a few specialized cases next. The kernel acquires a set of pages and creates page buffers that are used to store data. In this case, a page can simply be released from the buffers and made accessible for general use. This is an example of reclamation that does not require eviction. Another case arises when we use file-backed pages. A section of a file is mapped to the physical address space and the corresponding data is stored in memory. In this case, we can dynamically shrink the part of the file that is mapped to physical memory and release file-backed pages for general use. This is an example of page reclamation where there is no eviction.

Shrinking the memory footprint thus involves a variety of mechanisms. We can have plain old-fashioned eviction, or we can reclaim pages from specialized page buffers. The sizes of the latter can be adjusted dynamically to release pages and make them available to other processes.

> **Point 6.3.3**
>
> A page is said to be young if its recently accessed and its access bit is 1. Otherwise, it is said to be old.

The Need to Age

Let us understand the aging process in the MGLRU algorithm. As we have discussed earlier, the sequence number indicates the generation. The smaller is the sequence number (generation), the earlier is the generation.

The youngest generation number is `lrugen->max_seq`. The oldest generation numbers are `lrugen->min_seq[ANON]` (anonymous pages) and `lrugen->min_seq[FILE]` (file-backed pages), respectively. Note that we are differentiating between them because we wish to treat them differently. Next, we find the number of generations that are currently alive for a given type of pages.

First, assume that we are considering `ANON` pages and its minimum sequence number is `min_seq`. This means that the number of generations currently in use is `max_seq` - `min_seq` + 1. If the number of sequence numbers is less than a threshold `MIN_NR_GENS`, then we have too few active generations. We need to increase the number of sequence numbers in use such that we have enough generations in the system. If that does not happen, we will have too few generations and the information that we hold will be very coarse-grained. Hence, in this case, we increment `max_seq` by 1. It is important to note that by incrementing `max_seq` we are automatically increasing the distance between an existing sequence number (generation) and the current value of `max_seq`. This *automatically ages* the current pages by 1 generation. In fact, this is a neat trick for aging all pages without actually modifying their data !!! All that matters is the relative motion between the current sequence number and `max_seq` – instead of decrementing the former, we increment the latter.

Conversely, we also need to have a maximum cap on the number of generations we maintain. Let us name the upper threshold `MAX_NR_GENS`. Clearly `MAX_NR_GENS > MIN_NR_GENS`. If it is very large, then it means that the generation information is too fine-grained, and thus it is of limited use.

Listing 6.16: The aging algorithm
source : mm/vmscan.c#L4468

```
1  /* There are too few generations. Aging is required */
2  if (min_seq + MIN_NR_GENS > max_seq) return true;
3
4  ...
5
6  /* There are too many generations. Aging is not required */
7  if (min_seq + MIN_NR_GENS < max_seq) return false;
8
9  /* Come here only if min_seq + MIN_NR_GENS == max_seq */
10 /* The aging logic */
11 if (max_gen_pages * MIN_NR_GENS > total)
12     return true;
13 if (min_gen_pages * (MIN_NR_GENS + 2) < total)
14     return true;
15
16 /* default */
17 return false;
```

Let us first understand, whether we need to run aging or not in the first place. The logic is shown in Listing 6.16. The aim is to maintain the following relationship: `min_seq` + `MIN_NR_GENS` == `max_seq`. This means that we wish to

ideally maintain `MIN_NR_GENS+1` sequence numbers (generations). The check in Line 2 checks if there are too few generations. Then, definitely there is a need to run the aging algorithm. On similar lines, if the check in Line 7 is true, then it means that there are too many generations. There is no need to run the aging algorithm.

Next, let us consider the corner case when there is equality. First, let us define what it means for a page to be `max_gen_pages` or `min_gen_pages`[2]. A page is said to be `max_gen_pages` if its associated sequence number is equal to `max_seq`. This means that it belongs to the latest generation. On similar lines, a page is said to be `min_gen_pages` if its sequence number follows this relationship: `seq + MIN_NR_GENS == max_seq`. Given that we would ideally like to maintain the number of generations at `MIN_NR_GENS+1`, we track two important pieces of information – the number of `max_gen_pages` and `min_gen_pages` pages, respectively. The important thing to keep in mind is that $(\mid$ `max_gen_pages` $\mid + \mid$ `min_gen_pages` $\mid)$ is not equal to the number of pages in the system. They represent two ends of the range of sequence numbers.

The first check `max_gen_pages` $\times$ `MIN_NR_GENS` $>$ `total` ensures that if there are too many `max_gen_pages` pages, there is a need to run aging. The reason is obvious. We want to maintain a balance between `max_gen_pages` and `min_gen_pages` pages. Let us consider the next inequality: `min_gen_pages` $\times$ (`MIN_NR_GENS` $+ 2$) $<$ `total`. This clearly says that if the number of `min_gen_pages` pages is lower than what is expected (too few), then also we need to age to distribute the rest of the pages better across generations. An astute reader may notice that here that we add an offset of 2, whereas we did not add such an offset in the case of `max_gen_pages` pages. There is an interesting explanation here, which will help us appreciate the nuances involved in designing practical systems.

Trivia 6.3.1

As mentioned before, we wish to ideally maintain only `MIN_NR_GENS+1` generations. However, we also want to be conservative with aging. We don't want to do aging frequently. This is because aging does two things. The first is that we increment `max_seq`, which is a simple operation. Subsequently, there is a need to scan pages and start promoting a few to newer generations. This process also marks used pages as unused. Hence, the subsequent scanning process finds pages to evict and ends up increasing evictions. It is alright if the number of `max_gen_pages` pages is slightly more than `total/MIN_NR`$_G$`ENS + 1`, which is the ideal value. Over time some of them will get marked as unused. We do not want to run aging very frequently. Hence, the multiplier is set to `MIN_NR_GENS`. In the case of `min_gen_pages` pages, the safety margin works in the reverse direction. We can allow the number of `min_gen_pages` pages to be as low as `total / (MIN_NR_GENS + 2)`. This is because, we do not want to age too frequently, and in this case aging will cause `min_gen_pages` pages to get evicted. We would also like to reduce unnecessary eviction. Hence, we set the safety margin differently in this case.

---

[2]Note that the names of the variables are `young` and `old` (resp.) in code. They have been renamed to avoid confusion with the terms "young" and "old" in the text.

The Aging Process

Let us now understand the aging process in detail. We walk through all the page tables and find pages to age. This involves a walk through all the page tables of processes that are currently alive (`mm_struct` structures). It is possible to accelerate this process. Let us consider a PMD (page middle directory) that contains a set of 512 page table entries. We can skip it altogether if its page tables have not been accessed. The entire PMD can be marked to be unaccessed, in this case. A Bloom filter is used for this purpose.

A Bloom filter stores a set of PMD addresses. If a PMD address is found in the Bloom filter, then it **most likely contains young pages (recently accessed)**, or there is a false positive. The PMD is said to be young in this case. In the case of a false positive, it means that the PMD predominantly contains unaccessed pages – it was falsely flagged as recently accessed (or young). The PMD is actually not young. To eliminate this possibility, there is therefore a need to scan all the PMD's constituent pages and check if the pages were recently accessed or not. The young/old status of the PMD can then be accurately determined. This entails extra work. However, if a PMD address is not found, then it means that it predominantly contains old pages for sure (the PMD is old). Given that Bloom filters do not produce false negatives, we can skip such PMDs with certainty because they are old.

For the rest of the young pages, we clear the accessed bit and set the generation of the page or folio to `max_seq`. Automatically, the rest of the pages get aged.

Overview of the Eviction Algorithm

Let us look at the algorithm to evict folios (`evict_folios` function). This algorithm is invoked when there is a need to reduce the pressure on physical memory. It could be that we are trying to start new processes or allocate more memory to existing processes, and we find that there is not enough space in physical memory. It could also be that a process is trying to access a large file and a large part of the file needs to be *mapped* to memory (discussed in Chapter 7), and there is no memory space. In this case, it is important to create some space in memory by evicting a few folios that have *aged*. This is where the notion of having multiple generations is useful – we simply <u>evict</u> the earliest generation.

Any eviction algorithm takes the LRU state (`struct lruvec`) as input and a factor called `swappiness`. The factor `swappiness` determines the priority of the eviction algorithm. If this integer factor is close to 1, then it means that the algorithm gives priority to evicting file-backed pages. On the other hand, if it is closer to its maximum value 200, then it means that evicting anonymous pages has the highest priority. We shall continue to see this conflict between evicting anonymous and file-backed pages throughout this discussion. We always need to make a choice between the two.

Let us thus answer this question first and look at the relevant part of the kernel code. If no value for the `swappiness` is specified and, it is equal to 0, then we choose to evict file-backed pages. This is the default setting in this code. The rationale is as follows. In the case of anonymous pages, it is necessary to allocate a page in the swap space and write it back upon an eviction. However,

a file-backed page has a natural home on the storage device (the hard disk or SSD in most cases). There is no need for additional swap space management. Hence, in many cases, we may prefer to evict file-backed pages especially if we are not reading important file-backed data.

This is one argument, however, there is another reverse argument that says that evicting anonymous pages should have a higher priority. The rationale here is that file-backed pages are often code pages (backed by the program's binary file and DLLs) or important pages that contain data that the program requires. On the other hand, anonymous pages may sometimes not contain valid data especially if it corresponds to parts of the heap or stack that do not have data that is currently in use (it may very well be invalid because the stack has been rolled back).

Both the arguments are correct in different settings. Hence, Linux provides both the options. If nothing is specified, then the first argument holds – file-backed pages. However, if there is more information and the value of the `swappiness` variable is in the range 1-200, then we have tunable priorities.

Kernel Code

In the context of this discussion, let us look at the relevant kernel code in Listing 6.17. If `swappiness` is 0, then we choose the default option and decide to evict file-backed pages. Otherwise, we compare the minimum sequence numbers for both the types: file and anon. If anonymous pages have a lower generation, then it means that they are more aged, and thus should be evicted. However, if the reverse is the case – file-backed pages have a lower generation (sequence number) – then we don't evict them outright. We are slightly biased towards them.

Listing 6.17: The algorithm that chooses the type of the pages/folios to evict
source : mm/vmscan.c#L4974

```
1  if (!swappiness)  /* default */
2      type = LRU_GEN_FILE;
3  else if (min_seq[LRU_GEN_ANON] < min_seq[LRU_GEN_FILE])
4      type = LRU_GEN_ANON;
5  else if (swappiness == 1)
6      type = LRU_GEN_FILE;
7  else if (swappiness == 200)
8      type = LRU_GEN_ANON;
9  else if (!(sc->gfp_mask & __GFP_IO)) /* I/O operations are
       involved */
10     type = LRU_GEN_FILE;
11 else
12     type = get_type_to_scan(lruvec, swappiness, &tier);
```

Next, we look at the value of the `swappiness`. If it is 1 (minimum value), then file-backed pages are evicted. Conversely, if it is 200, then anonymous pages are evicted. These are extrema of the spectrum.

Next, we check if I/O operations will be required if we swap a page out such as writing to the disk; we check whether the flag `__GFP_IO` is true or not (GFP stands for Get Free Pages). In general, we require them, where I/O operations are required to write the evicted page to swap space. However, in some contexts such as atomic contexts, we would like to disallow this because evicting anonymous pages when I/O operations are underway can cause correctness problems.

In this case, we should evict file-backed pages. Given that they are already backed up, eviction is a more seamless process.

If none of these conditions hold, it is necessary to find the type using a more elaborate algorithm. We thus invoke the `get_type_to_scan` function (scan pages for eviction).

Finding the Type of Pages to Evict

We use a control- theory-inspired algorithm to balance the eviction of file-backed and anonymous pages. We define the `ctrl_pos` structure that contains the following pieces of information: the total number of page faults, the number of refaults and a quantity called the `gain`. A refault is an access to an evicted page, which is not a desirable thing. We would ideally want the refault rate to be as close to zero as possible, which means that we would not like an evicted page to be accessed again (not in the near future at least).

Next, we define two variables namely `sp` (set point) and `pv` (process variable). The goal of any such algorithm inspired by control theory is to set the process variable equal to the set point (a pre-determined state of the system). We shall stick to this generic terminology because such an algorithm is valuable in many scenarios, not just in finding the type of pages to evict. It tries to bring two quantities closer in the real world. Hence, we would like to explain it in general terms.

The parameter `gain` plays an important role here. It sets the aggressiveness of the system in determining the output. The output divided by the gain is the normalized output. Across systems, typically there is a need to equalize this. This means that if the gain is high, we can afford a higher output and vice versa. Let us explain. For anonymous pages, `gain` is defined as the `swappiness` (higher it is, more are the evicted anonymous pages) and for file-backed pages, it is `200-swappiness`. `gain` indirectly is a measure of the effort invested in trying to evict a given type of pages. If it approaches 200, then we wish to only evict *anon* pages, and if it approaches 1, we wish to only evict *file* pages. The refault rate divided by the gain is the mean refault rate achieved per unit effort. We would like this to be equal for *file* and *anon* pages.

Next, we initialize the `sp` and `pv` variables. The set point is set equal to the eviction statistics ⟨ # refaults, #evictions, gain ⟩ of *anon* pages that are *clean* (not accessed after the stats were reset). The process variable is set to the eviction statistics of clean *file* pages. We need to now compare `sp` and `pv`. Note that we are treating `sp` (*anon*) as the reference point here and trying to ensure that "in some sense" `pv` approaches `pv` (*file*) approaches `sp`. This will balance both of them, and we will be equally fair to both of them.

> **Point 6.3.4**
>
> In any control-theoretic algorithm, our main aim is to bring `pv` as close to `sp` as possible. In this case also we wish to do so and in the process ensure that both *file* and *anon* pages are treated *fairly*.

Let us do a case-by-case analysis and understand the rationale behind what Linux does. First, we should avoid making decisions based on the absolute number of *file* or *anon* refaults because the refault rate might be high, but the total number of evictions could also be high. Hence, it is a wiser idea to actually

look at the ratio $(\#refaults)/(\#evictions)$ separately for $file$ and $anon$ pages. Let us refer to this ratio as the *normalized refault rate*. We should then compute a ratio of the ratios, or in other words a ratio of the normalized refault rates. This makes somewhat more sense, because we are normalizing the number of refaults with the total number of evictions. If this ratio of ratios ($file/anon$) is relatively low, then clearly the normalized refault rate for $file$ pages is low, and thus it can be increased further by evicting a few $file$ pages, and vice versa. This will enhance fairness.

Note that we have not taken the `swappiness` into account yet. It determines the `gain` for $file$ and $anon$ pages. As discussed before, the refault rate or normalized refault rate divided by the `gain` is a measure of how much the mean refault rate could be affected per unit effort. We would like this to be the same for $file$ and $anon$ pages. This means that the system is not over or under-responsive.

$$\frac{\texttt{pv.refaulted}}{\texttt{pv.total} \times \texttt{pv.gain}} \leq \frac{\texttt{sp.refaulted}}{\texttt{sp.total} \times \texttt{sp.gain}} \tag{6.3}$$

This rationale is captured in Equation 6.3. If the normalized refault rate of pv divided by its gain is less than the corresponding quantity of the set point, then we modify the pv. In other words, we choose pages to evict from the pv class ($file$ class).

$$\frac{\texttt{pv.refaulted}}{\texttt{pv.total} \times \texttt{pv.gain}} \leq \frac{(\texttt{sp.refaulted} + \alpha)}{(\texttt{sp.total} + \beta) \times \texttt{sp.gain}} \tag{6.4}$$

Equation 6.3 is an ideal equation. In practice, Linux adds a couple of constants to the numerator and the denominator to incorporate practical considerations and maximize the performance of real workloads. Hence, the exact version of the formula implemented in the Linux kernel v6.2 is shown in Equation 6.4. The designers of Linux set $\alpha = 1$ and $\beta = 64$. Note that these constants are based on experimental results, and it is hard to explain them logically.

There is another small trick. If the absolute value of the $file$ refaults is low < 64, then MGLRU chooses to evict file-backed pages. The reason is that most likely the application either does not access a file or the file accessed is very small. On the other hand, every application shall have a large amount of anonymous memory comprising stack and heap sections. Even if it is a small application, the code to initialize it is sizable. Hence, it is good idea to evict $anon$ pages only if the $file$ refault rate is above a certain threshold.

Now, we clearly have a winner. If the inequality in Equation 6.4 is true, then pv is chosen ($file$), else sp is chosen ($anon$).

Scan the Folios

We iterate through all the zones that need to be shrunk. In each zone, we scan all its constituent folios. The idea is to scan all the folios with the least sequence number of the type of pages that need to be evicted. The structure `lrugen` maintains such a list (see Listing 6.13).

Given a folio, we check whether it can be evicted or not. If the folio is pinned, is in the process of being written back, was recently accessed or there is a possibility of a race condition, then it is best to skip it. The folio should not

be evicted at least till these conditions are true. The end result of this process is the number and list of folios that can possibly be evicted.

There is a question of fairness here. Assume that our algorithm selects *anon* folios to be evicted. It is true that they are all from the lowest generation. However, it would be unfair to evict all of them. Some of them may have been accessed a lot in the past. There is thus a need for doing some degree of filtration here and identifying those folios for eviction that appear to be "genuine" candidates. There is a need to split a generation into multiple *tiers* where different tiers can be treated differently.

Tiers

Let us now delve into the notion of *tiers*. Let us count the number of references to folios. A reference in this case is approximately counted, where the count is incremented when there is a page fault (hard or soft), or we find the page has been accessed on a page table scan. This reference count is indicative of the frequency of accesses to pages within the folio. The tier is set to $\lceil log(refs+1) \rceil$, where $refs$ is the number of references to the folio. Folios in lower tiers are infrequently referenced, while folios in higher tiers are more frequently referenced. We would ideally like to evict folios in lower tiers. Folios in higher tiers may be more useful. Recall that in Equation 6.4, we were comparing the statistics for *clean* pages. They are in tier 0.

Assume that the type of pages chosen for eviction is T, and the other type (not chosen for eviction) is T'. The choice was made based on statistics for tier 0. Let us now do a tier-wise analysis, and compare their normalized refault rates by taking the `gain` into account tier-wise. We shall use Equation 6.4. We will compare the normalized refault rates of all the tiers of T with tier 0 of T'.

We may find that till a certain tier k, folios of type T need to be evicted because their normalized refault rate divided by the gain is comparatively lower. Given Equation 6.4 and the way in which we chose type T, we know that this holds for tier 0 for sure. Let this property hold till tier k. Now, for tier $k+1$ (≥ 1), the reverse is the case. It means that folios of type T' need to be evicted as per the logic in Equation 6.4. If no such k exists, then nothing needs to be done. Let us consider the case, when we find such a value of k.
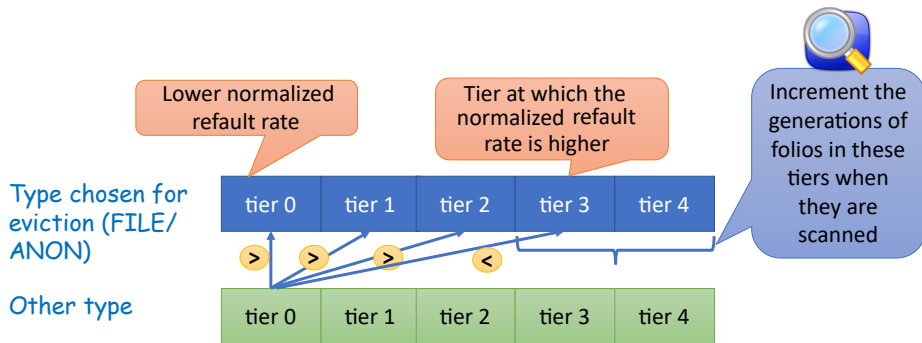
Figure 6.24: The notion of tiers

In this case, the folios in tiers $[0, k]$ of type T should be considered for eviction as long as they are not pinned, being written back, involved in race conditions,

etc. However, the folios in tiers $k + 1$ and beyond, should be given a second chance because they have seen more references and have a comparatively higher refault rate. If we compare tier-specific statistics, it is clear that by Equation 6.4 they should remain in memory. Instead of doing the same computation for the rest of the folios in other tiers, we can simply assume that they also need to be kept in memory for the time being. This is done in the interest of time and there is a high probability of this being a good decision. Note that higher-numbered tiers see an exponential number of more references. Hence, we increment the generations of all folios in the range $[k+1, MAX\_TIERS]$. They do not belong to the oldest generation anymore. They enjoy a second chance. Once a folio is promoted to a new generation, we can clear its reference count. The philosophy here is that the folio gained because of its high reference count once. Let it not benefit once again. Let it start afresh after getting *promoted* to the higher generation. This is depicted pictorially in Figure 6.24.

Eviction of a Folio

Now, we finally have a list of folios that can be evicted. Note that some folios did get rescued and were given a second chance.

The process of eviction can now be started folio after folio. Sometimes it is necessary to insert short delays in this process, particularly if there are high-priority tasks that need to access storage devices or some pages in the folio are being swapped out. This is not a one-shot process, it is punctuated with periods of activity initiated by other processes.

Once a folio starts getting evicted, we can do some additional bookkeeping. We can scan proximate (nearby) virtual addresses. The idea here is that programs tend to exhibit spatial locality. If one folio was found to be old, then pages in the same vicinity should also be scrutinized. We may find many more candidates that can possibly be evicted in the near future. For such candidate pages, we can mark them to be *old* (clear the accessed bit) and also record the PMD (Page Middle Directory) entries that comprise mostly of *young* pages. These can be added to a Bloom filter, which will prove to be very useful later in the page walk process. We can also slightly reorganize the folios here. If a folio is very large, it can be split to several smaller folios.

> **Point 6.3.5**
>
> Such additional bookkeeping actions that are piggybacked on regular operations like a folio eviction are a common pattern in modern operating systems. Instead of operating on large data structures like the page table in one go, it is much better to slightly burden each operation with a small amount of additional bookkeeping work. For example, folio eviction is not on the critical path most of the time, and thus we can afford to do some extra work.

Once the extra work of bookkeeping is done, the folio can be written back to the storage device. This would involve clearing its kernel state, freeing all the buffers that were storing its data (like the page cache for file-backed pages), flushing the relevant entries in the TLB and finally writing the folio back.

The Process of Looking Around

In the previous subsection, we had mentioned that while evicting a folio, the MGLRU algorithm also looks at nearby pages and marks them to be *old*. Let us look at this process in some more detail (refer to the function `lru_gen_look_around` in the kernel code).

First, note that this function can be invoked in many different ways. We have already seen one, which is at the time of evicting a folio. However, it can also be invoked by the kernel daemon `kswapd`, which is a kernel process that runs periodically; it tries to age and evict pages. Its job is to run in the background as a low-priority process and clear pages from memory that are unlikely to be used in the near future. Its job is to walk through the page tables of processes, mark *young* pages as *old* (inspired by the clock-based page replacement algorithms) and increment the generation of *young entries*. Note that this process only considers *young* (recently accessed) pages. Old entries are skipped. A fast and performance-efficient way of doing this is as follows.

We enter PMD addresses (2nd lowest level of the page table) in a Bloom filter if they predominantly contain *young* pages. Now, we know that in a Bloom filter, there is no chance of a false negative. This means that if it says that a given PMD address is not there, it is not there for sure. Subsequently, when we walk the page table, we query the Bloom filter to check if it has a given PMD address. If the answer is in the negative, then we know that it is correct, and it is not there because it contains mostly *old* pages. Given that there is no possibility of an error, we can confidently skip scanning all the constituent page table entries that are covered by the PMD entry. This will save us a lot of time.

Let us consider the other case, when we find a PMD address in the Bloom filter. It is most likely dominated by *young pages*. The reason we use the term "most likely" because Bloom filters can lead to false positive outcomes. We scan the pages in the PMD region – either all or a subset of them at a time based on performance considerations. This process of looking around marks *young* folios as *old* on the lines of classic clock-based page replacement algorithms. Moreover, note that when a folio is marked, all its constituent pages are also marked. At this point, we can do some additional things. If we find a PMD region to comprise mostly of *young* pages, then the PMD address can be added to the Bloom filter. Furthermore, *young* folios in this region can be promoted to the latest generation – their generation/sequence number can be set to `max_seq`. This is because they are themselves *young*, and they also lie in a region that mostly comprises *young* pages. We can use spatial and temporal locality based arguments to justify this choice.

6.3.3 Thrashing

Your author is pretty sure that everybody is guilty of the following performance crime. The user boots the machine and tries to check her email. She finds it to be very slow because the system is booting up and all the pages of the email client are not in memory. She grows impatient, and tries to start the web browser as well. Even that is slow. She grows even more impatient and tries to write a document using MS Word. Things just keep getting slower. Ultimately, she gives up and waits. After a minute or two, all the applications come up and the system stabilizes. Sometimes if she is unlucky, the system crashes.

What exactly is happening here? Let us look at it from the point of view of paging. Loading the pages for the first time into memory from a storage device such as a hard disk or even a flash drive takes time. Storage is several orders of magnitude slower than main memory. During this time, if another application is started, its pages also start getting loaded. This reduces the bandwidth of the storage device and both applications get slowed down. However, this is not the only problem. If these are large programs, whose working set (refer to Section 6.1.3) is close to the size of main memory, then they need to evict each other's pages. As a result, when we start a new application, it evicts pages of the applications that are already running. Then, when there is a context switch, existing applications stall because crucial pages from their working set were evicted. They suffer from page faults. Their pages are then fetched from memory. However, this has the same effect again. These pages displace the pages of other applications. This cycle continues. This phenomenon is known as *thrashing*. A system goes into thrashing when there are too many applications running at the same time and most of them require a large amount of memory. They end up evicting pages from each other's working sets, which just increases the page fault rate without any beneficial outcome.

It turns out that things can get even worse. The performance counters detect that there is low CPU activity. This is because most of the time is going in servicing page faults. As a result, the scheduler tries to schedule even more applications to increase the CPU load. This increases the thrashing even further. This can lead to a vicious cycle, which is why thrashing needs to be detected and avoided at all costs.

Linux has a pretty direct solution to stop thrashing. It tries to keep the working set of an application in memory. This means that once a page is brought in, it is not evicted very easily. The page that is brought in (upon a page fault) is most likely a part of the working set. Hence, it makes little sense to evict it. The MGLRU algorithm already ensures this to some extent. A page that is brought into main memory has the latest generation. It takes time for it to age and be a part of the oldest generation and become eligible for eviction. However, when there are a lot of applications, the code in Listing 6.16 can trigger the aging process relatively quickly because we will just have a lot of `max_gen_pages` pages. This is not a bad thing when there is no thrashing. We are basically weeding out `min_gen_pages` pages. However, when thrashing sets in, such mechanisms can behave in erratic ways.

There is thus a need for a master control. The eviction algorithm simply does not allow a page to be evicted if it was brought into memory in the last N ms. In most practical implementations, $N = 1000$. This means that every page is kept in memory for at least 1 second. This ensures that evicting pages in the working set of any process is difficult. Thrashing can be effectively prevented in this manner.

However, there is one problem here. Assume that an application is trying to execute, but its pages cannot be loaded to memory because of the aforementioned rule. In this case, it may wait in the runqueue for a long time. This will make it unresponsive. To prevent this, Linux simply denies it permission to run and terminates it with an "Out of Memory" (OOM) error. It has a dedicated utility called the OOM killer whose job is to terminate such applications. This is a form of admission control where we limit the number of processes. Along with persisting working set pages in memory for a longer duration, terminating

new processes effectively prevents thrashing.

Definition 6.3.1 Thrashing

Thrashing refers to a phenomenon where we run too many applications and all their working sets do not fit in memory. They thus end up evicting pages from each other's working sets on a continual basis and most of the time their execution is stalled. The scheduler may sense that the CPU load is low and further schedule more processes, which exacerbates the problem. This vicious cycle has the potential to continue and ultimately bring down the system.

6.4 Kernel Memory Allocation

Let us now discuss kernel memory allocation, which is quite different from memory allocation schemes in the user space. We have solved almost all user-level problems using virtual memory and paging. We further added some structure to the user-level virtual address space. Recall that every user process has a virtual memory map where the virtual address is divided into multiple sections such as the stack, heap, text section, etc.

We had also discussed the organization of the kernel's virtual address space in Section 6.2.1. Here we saw many regions that are either not "paged", or where the address translation is a simple linear function. This implies that contiguity in the virtual address implies contiguity in the physical address space as well. We had argued that this is indeed a very desirable feature especially when we are communicating with external I/O devices, DMA controllers and managing the memory space associated with kernel-specific structures. Having some control over the physical memory space was deemed to be a good thing.

On the flip side, this will take us back to the bad old days of managing a large chunk of contiguous memory without the assistance of paging-based systems that totally delink the virtual and physical address spaces, respectively. We may again start seeing a fair amount of external fragmentation. Notwithstanding this concern, we also realize that in paging systems, there is often a need to allocate a large chunk of contiguous physical addresses. This is quite beneficial because prefetching-related optimizations are possible. In either case, we are looking at the same problem, which is maintaining a large chunk of contiguous memory while avoiding the obvious pitfalls: management of holes and uncontrolled external fragmentation

Recall that we had discussed the base-limit scheme in Section 6.1.1. It was solving a similar problem, albeit ineffectively. We had come across the problem of holes, and it was very difficult to plug holes or solve the issues surrounding external fragmentation. We had proposed a bunch of heuristics such as first-fit, next-fit and so on, however we could not come up with a very effective method of managing the memory this way. It turns out that if we have a bit more of regularity in the memory accesses, then we can use many other ingenious mechanisms to manage the memory better without resorting to conventional paging. We will discuss several such mechanisms in this section.

6.4.1 Buddy Allocator

Let us start with discussing one of the most popular mechanisms for kernel memory allocation namely *buddy allocation*. It is often used for managing physical memory, however as we have discussed such schemes are useful for managing any kind of contiguous memory including the virtual address space. Hence, without looking at the specific use case, let us look at the properties of the allocator where the assumption is that the addresses are contiguous (most often physical, sometimes virtual).
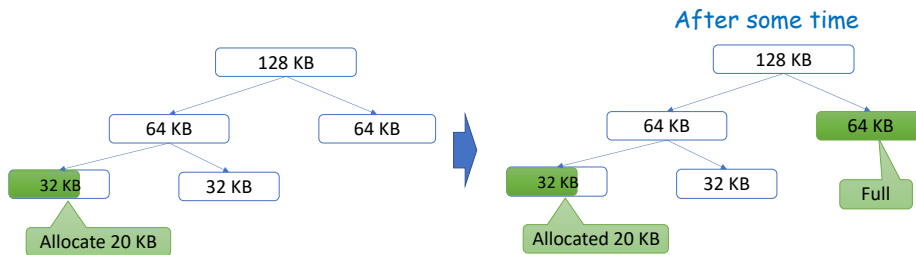
Figure 6.25: Buddy allocation

The concept of a buddy is shown in Figure 6.25. Let us explain the key idea informally first. Consider a region of memory whose size in bytes or kilobytes is a power of 2. For example, in Figure 6.25, we consider a 128 KB region. Assume that we need to make an allocation of 20 KB. We split the 128 KB region into two regions that are 64 KB each. They are said to be the *buddies* of each other. Then we split the left 64 KB region into two 32 KB regions, which are again buddies of each other. Now we can clearly see that 20 KB is between two powers of two: 16 KB and 32 KB. Hence, we take the leftmost 32 KB region and out of that we allocate 20 KB to the current request. We basically split a large free region into two equal-sized smaller regions until the request lies between the region size and the region size divided by two. We are basically overlaying a binary tree on top of a linear array of pages.

If we traverse the leaves of this buddy tree from left to right, then they essentially form a partition of the single large region. An allocation can only be made at the leaves. If the request size is less than half the size of a leaf node that is unallocated, then we split it into two equal-sized regions (contiguous in memory), and continue to do so until we can just about fit the request. Note that throughout this process, the size of each subregion is still a power of 2.

Now assume that after some time, we get a request for a 64 KB block of memory. Then as shown in the second part of Figure 6.25, we allocate the remaining 64 KB region (right child of the parent) to the request.

Let us now free the 20 KB region that was allocated earlier (see Figure 6.26). In this case, we will have two 32 KB regions that are free and next to each other (they are siblings in the tree). There is no reason to have two free regions at the same level. Instead, we can get rid of them and just keep the parent, whose size is 64 KB. We are essentially merging free regions (holes) and creating a larger free region. In other words, we can say that if both the children of a parent node are free (unallocated), they should be removed, and we should only have the parent node that coalesces the full region. The parent now becomes a leaf.
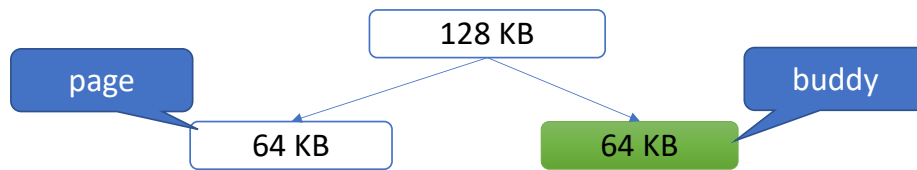
Figure 6.26: Freeing the 20 KB region allocated earlier

Let us now look at the implementation. We refer to the region represented by each node in the buddy tree as a *block*. Its minimum granularity of storage is one page.

Implementation

We start by revisiting the `free_area` array in `struct zone` (refer to Section 6.2.5). We shall define the *order* of a node in the buddy tree. The order of a leaf node that corresponds to the smallest possible region – one page – is 0. Its parent has order 1. The order keeps increasing by 1 till we reach the root. Let us now represent the tree as an array of lists: one list per order. All the *free* (unallocated) nodes of the tree (of the same order) are stored one after the other (left to right) in an order-specific list. A *node* represents an aggregate page, which stores a block of memory depending upon the order. Thus, we can say that each linked list is a list of compound pages, where each compound page is actually an aggregate page that may point to N contiguous 4 KB pages, where N is a power of 2. Note that each such compound page needs to be completely unallocated for it to be a part of the list.

The buddy tree is thus represented by an array of linked lists – `struct free_area free_area[MAX_ORDER]`. Refer to Listing 6.18, where each `struct free_area` is a linked list of free nodes (of the same order). The root's order is `MAX_ORDER - 1`. In each `free_area` structure, the member `nr_free` refers to the number of free blocks.

There is a subtle twist involved here. We actually have multiple linked lists – one for each *migration type*. The Linux kernel classifies pages based on their migration type: it is based on whether they can move, once they have been allocated. One class of pages cannot move after allocation, then there are pages that can freely move around physical memory, there are pages that can be reclaimed and there are pages reserved for specific purposes. These are different examples of migration types. We maintain separate lists for different migration types. It is as if their memory is managed separately.

Listing 6.18: `struct free_area`
source : include/linux/mmzone.h#L105

```
struct zone {
    ...
    struct free_area  free_area[MAX_ORDER];
    ...
}
struct free_area {
    /* unmovable, movable, reclaimable, ...*/
```

```
    struct list_head   free_list[MIGRATE_TYPES];
    unsigned long      nr_free;
};
```

The take-home point is that a binary tree is represented as an array of lists
– one list for each order. Each node in a linked list is an aggregate or compound
page. The buddy tree is an integral part of a zone – it is its default memory
allocator.
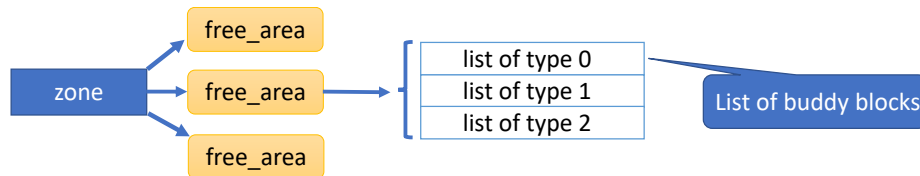
Figure 6.27: Buddies within a zone. The *type* refers to the migration type

We can visualize this organization in Figure 6.27, where we see that a zone
has a pointer to a single `free_area` (for a given order), and this `free_area` struc-
ture has pointers to many lists depending on the type of the page migration that
is allowed. Each list contains a list of free blocks (aggregate pages). Effectively,
we are maintaining multiple buddy trees – one for each page reclamation type.

An astute reader may ask how the buddy tree is being created – there are
after all no parent or child pointers. This is *implicit*. We will soon show that
parent-child relationships can be figured out with simple pointer arithmetic.
There is a no need to store pointers. Keep reading.

Kernel Code for Allocating/Freeing Blocks

Block Allocation

Listing 6.19 shows the relevant kernel code for allocating a block. Here, we
traverse a list that holds `free_areas`. The aim is to find the first available
block for memory allocation. We start with a simple *for* loop that traverses the
tree from a given order to the highest (towards the root). At each level, we
find a relevant `free_area` (`area`). For a given migration type, we try to get a
pointer to a block (stored as an aggregate `page`). If a block is not found, which
basically means that either there is no free block or the size of the request is
more than the size of the block, then we continue iterating and increase the
order. This basically means that we go towards the root. However, if a block
of appropriate size is found, then we delete it from the free list by calling the
`del_page_from_free_list` function and return it.

Listing 6.19:   Traversing   the   list   of   `free_area` s   in   the   function
`__rmqueue_smallest`
source : mm/page_alloc.c#L2554

```
for (current_order = order; current_order < MAX_ORDER; ++
    current_order) {
    area = &(zone->free_area[current_order]);
    page = get_page_from_free_area(area, migratetype);
```

```
    if (!page)
        continue;

    /* block found */
    del_page_from_free_list(page, zone, current_order);
    ...
    return page;
}
```

Freeing a Block

Listing 6.20 shows the code for freeing an aggregate page (block in the buddy system). In this case, we start from the block that we want to free and keep proceeding towards the root. Given a page, we find the page frame number of the buddy. If the buddy is not free then the `find_buddy_page_pfn` function returns NULL. Then, we exit the *for* loop and go to label `done_merging`. Nothing more needs to be done. If this is not the case, we delete the buddy and coalesce the page with the buddy.

Let us explain this mathematically. Assume that pages with frame numbers A and B are buddies of each other. Let the order be ϕ. Without loss of generality, let us assume that $A < B$. Then we can say that $B = A + 2^\phi$, where $\phi = 0$ for the lowest level (the unit here is pages). Now, if we want to combine A and B and create one single block that is twice the block size of A and B, then it needs to start at A and its size needs to be $2^{\phi+1}$ pages.

Let us now remove the restriction that $A < B$. Let us just assume that they are just buddies of each other. We then have $A = B \oplus 2^\phi$. Here $\oplus$ stands for the XOR operator. Now, if we coalesce A and B, the aggregate page corresponding to the parent node needs to have its starting pfn (page frame number) at $min(A, B)$. This is the same as $A\&B$, where & stands for the logical AND operation. This is because they vary at a single bit: the $(\phi+1)^{th}$ bit (LSB is bit number 1). If we compute a logical AND, then this bit gets set to 0, and we get the minimum of the two pfns. Let us now compute $min(A, B) - A$. It can either be 0 or -2^ϕ, where the order is ϕ.

We implement exactly the same logic in Listing 6.20, where A and B are the `buddy_pfn` and `pfn`, respectively. The `combined_pfn` represents the minimum: starting address of the new aggregate page. The expression `combined_pfn - pfn` is the same as $min(A, B) - A$. If $A < B$, it is equal to 0, which means that the aggregate page (corresp. to the parent) starts at `struct page* page`. However, if $A > B$, then it starts at `page` minus an offset. The offset should be equal to $A - B$ multiplied by the size of `struct page`. In this case $A - B$ is equal to `pfn - combined_pfn`. The reason that this offset gets multiplied with `struct page` is because when we do pointer arithmetic in C, any constant that gets added or subtracted to a pointer automatically gets multiplied by the size of the structure (or data type) that the pointer is pointing to. In this case, the pointer is pointing to date of type `struct page`. Hence, the negative offset `combined_pfn - pfn` also gets multiplied with `sizeof(struct page)`. This is the starting address of the aggregate page (corresponding to the parent node).

Listing 6.20: Code for freeing a page
source : mm/page_alloc.c#L1092

```
void __free_one_page(struct page *page, unsigned long pfn,
```

```
    struct zone *zone, unsigned int order, ...)
{

    while (order < MAX_ORDER - 1) {
        buddy = find_buddy_page_pfn(page, pfn, order, &
            buddy_pfn);
        if (!buddy)
            goto done\_merging;
        del_page_from_free_list(buddy, zone, order);
        ....
        combined_pfn = buddy_pfn & pfn;
        page = page + (combined_pfn - pfn);
        pfn = combined_pfn;
        order++;
    }

done_merging:
    /* set the order of the new
    set_buddy_order(page, order);
    add_to_free_list(page, zone, order, migratetype);
}
```

The pointer arithmetic can be complex. We request the reader to manually work out a small example.

Once we combine a page and its buddy, we increment the order and try to combine the parent with its buddy, so on and so forth. This process continues until we are successful. Otherwise, we break from the loop and reach the label **done_merging**. Here we set the order of the merged (coalesced) page and add it to the free list at the corresponding order. This completes the process of freeing a node in the buddy tree.

Point 6.4.1

The buddy system overlays a possibly unbalanced binary tree over a linear array of pages. Each node of the tree corresponds to a set of contiguous pages (the number is a power of 2). The range of pages represented by a node is equally split between its children (left-half and right-half). This process continues recursively. The allocations are always made at the leaf nodes that are also constrained to have a capacity of N pages, where N is a power of 2. It is never the case that two children of the same node are free (unallocated). In this case, we need to delete them and make the parent a leaf node. Whenever an allocation is made at a leaf node, the allocated memory always exceeds 50% of the capacity of that node (otherwise we would have split that node). Note that there is a corner case here. We may want to allocate just 10 bytes. In this case, the leaf node has to contain just a single page.

6.4.2   Slab Allocator

Now that we have seen the buddy allocator, which is a generic allocator that manages contiguous sections of the kernel memory quite effectively, let us move to allocators for a single class of objects. Recall that we had discussed object

pools (`kmem_cache` s) in Section 3.1.12. If we were to create such a pool of objects, then we need to find a way of managing contiguous memory for storing a large number of objects that have exactly the same size. For such use cases, the slab allocator is quite useful. In fact, it is often used along with the buddy allocator. We can use the buddy allocator as the high-level allocator to manage the overall address space. It can do a high-level allocation and give a contiguous region to the slab allocator, which it can then manage on its own. It can use this memory region for creating its pool of objects (of a single type) and storing other associated data structures.

Let us now discuss the slab allocator in detail. As of kernel v6.2, it is the most popular allocator, and it has managed to make the earlier slob allocator obsolete. We will discuss the slab allocator and then a more optimized version of it – the Slub allocator.

The high-level diagram of the allocator is shown in Figure 6.28. The key concept here is that of a *slab*. It is a generic storage region that can store a set of objects of the same type. Assume that it can store k objects. Then the size of the memory region for storing objects is $k \times sizeof(object)$. A pointer to this region that stores objects is stored in the member `s_mem` of `struct slab`.

It is important to note that all these objects in this set of k objects may not actually be allocated and be active. It is just that space is reserved for them. Some of these objects may be allocated whereas the rest may be unallocated (or free). We can maintain a bit vector with k bits, where the ith bit is set if the ith object has been allocated and some task is using it. A slab uses a `freelist` to store the indexes of free objects. Every slab has a pointer to a slab cache (`kmem_cache`) that contains additional slabs.

Note that all of these entities such as a slab and slab cache are specific to only one object type. We need to define separate slabs and slab caches for each type of object that we want to store in a pool.
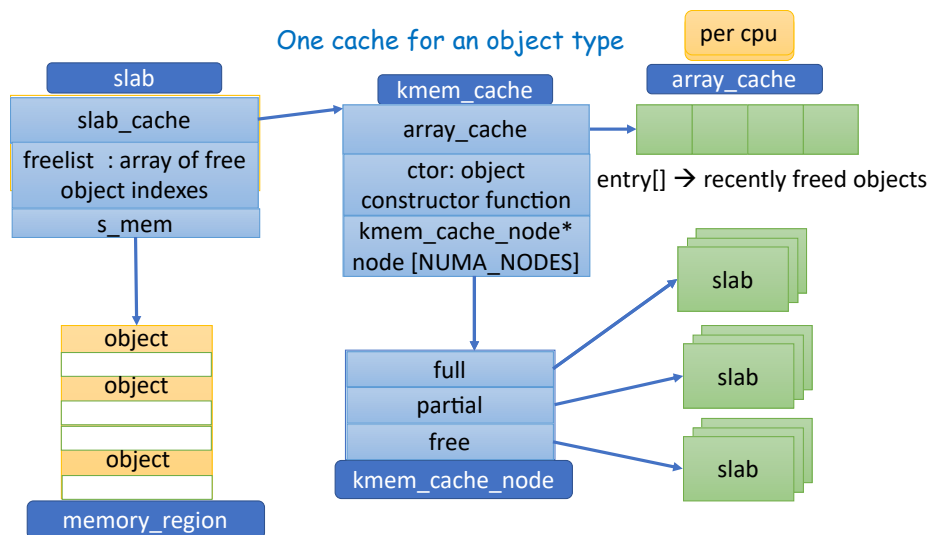
Figure 6.28: The slab allocator
mm/slab.h

The slab cache has a per-CPU array of free objects (`array_cache`). These are recently freed objects, which can be quickly reused. This is a very fast way of allocating an object without accessing other data structures to find which object is free. Every object in this array is associated with a slab. Sadly, when such an object is allocated or freed, the state in its encapsulating slab needs to also be changed. We will see later that this particular overhead is not there in the Slub allocator.

Now, if there is a high demand for objects, then we may run out of free objects in the per-CPU `array_cache`. In such a case, we need to find a slab that has a free object available.

It is very important to appreciate the relationship between a slab and the slab cache (`kmem_cache`) at this point of time. The slab cache is a system-wide pool whose job is to provide a free object and also take back an object after it has been used (added back to the pool). A slab on the other hand is just a storage area for storing a set of k objects: both active and inactive.

The slab cache maintains three kinds of slab lists – *full*, *partial* and *free* – for each NUMA node. The full list contains only slabs that do not have any free object. The partial list contains a set of partially full slabs and the free list contains a set of slabs that do not have a single allocated object. The algorithm is to first find a partially full slab. Then in that slab, it is possible to find an object that has not been allocated yet. The state of the object can then be initialized using an initialization function whose pointer must be provided by the user of the slab cache. The object is now ready for use.

However, if there are no partially full slabs, then one of the empty slabs needs to be taken and converted to a partially full slab by allocating an object within it.

We follow the reverse process when returning an object to the slab cache. Specifically, we add it to the `array_cache`, and set the state of the slab that the object is a part of. This can easily be found out by looking at the address of the object and then doing a little bit of pointer math to find the nearest slab boundary. If the slab was full, then now it is partially full. It needs to be removed from the full list and added to the partially full list. If this was the only allocated object in a partially full slab, then the slab is empty now.

We assume that a dedicated region in the kernel's memory map is used to store the slabs. Clearly all the slabs have to be in a contiguous region of the memory such that we can do simple pointer arithmetic to find the encapsulating slab. The memory region corresponding to the slabs and the slab cache can be allocated in bulk using the high-level buddy allocator.

This is a nice, flexible and rather elaborate way of managing physical memory for storing objects of only a particular type. A criticism of this approach is that there are too many lists, and we frequently need to move slabs from one list to the other.

6.4.3   Slub Allocator

The Slub allocator is comparatively simpler; it relies heavily on pointer arithmetic. Its structure is shown in Figure 6.29.

We reuse the same slab structure that was used in the slab allocator. We specifically make use of the `inuse` field to find the number of objects that are currently being used and the `freelist`. Note that we have compressed the slab
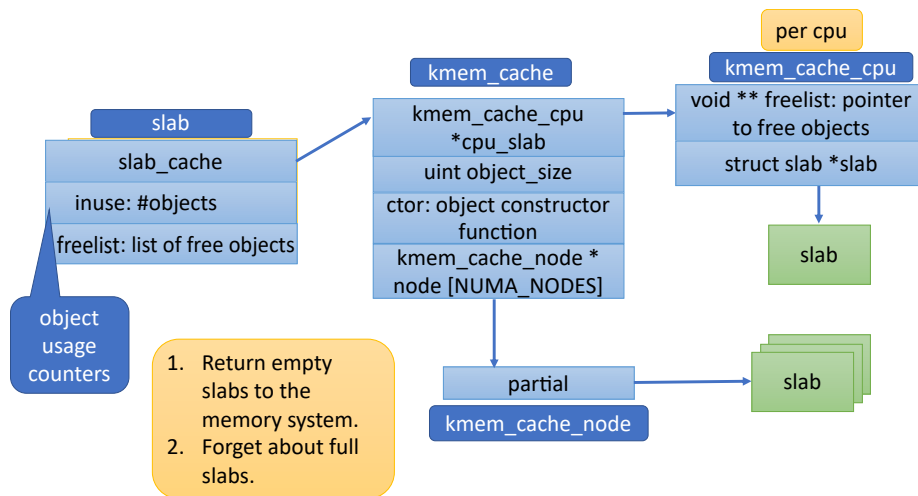
Figure 6.29: The Slub allocator

part in Figure 6.29 and just summarized it. This is because it has been shown in its full glory in Figure 6.28.

Here also every slab has a pointer to the slab cache (`kmem_cache`). However, the slab cache is architected differently. Every CPU in this case is given a private slab that is stored in its per-CPU region. We do not have a separate set of free objects for quick allocation. It is necessary to prioritize regularity for achieving better performance. Instead of having an array of recently-freed objects, a slab is the basic/atomic unit here. From the point of view of memory space usage and sheer simplicity, this is a good idea.

There are performance benefits because there is more per-CPU space, and it is quite easy to manage it. Recall that in the case of the slab allocator, we had to also go and modify the state of the slabs that encapsulated the allocated objects. Here we maintain state at only one place, and we never separate an object from its slab. All the changes are confined to a slab and there is no need to go and make changes at different places. We just deal in terms of slabs and assign them to the CPUs and slab caches at will. Given that a slab is never split into its constituent objects, their high-level management is quite straightforward.

If the per-CPU slab becomes full, all that we need to do in this case is simply forget about it and find a new free slab to assign to the CPU. In this case, we do not maintain a list of completely empty and full slabs. We just forget about them. We only maintain a list of partially full slabs, and query this list of partially full slabs, when we do not find enough objects in the per-CPU slab. The algorithm is the same. We find a partially full slab and allocate a free object. If the partially full slab becomes full, then we remove it from the list and forget about it. This makes the slab cache much smaller and more memory efficient. Let us now see where pointer math is used.

We need to do a good job without maintaining a list of full and empty slabs. If an object is deallocated, we need to return it back to the pool. From the object's address, we can figure out that it was part of a slab. This is because slabs are stored in a dedicated memory region. Hence, the address is sufficient

to figure out that the object is a part of a slab, and we can also find the starting address of the slab by computing the nearest "slab boundary". We can also figure out that the object is a part of a full slab because the slab is not present in the slab cache. Now that the object is being returned to the pool, a full slab becomes partially full. We can then add it to the list of partially full slabs in the slab cache.

Similarly, we do not maintain a list of empty slabs because there is no reason to do so. These empty slabs are returned to the buddy allocator such that they can be used for other purposes. Whenever there is a need for more slabs, they can be fetched on demand from the high-level buddy allocator. Subsequently, it can be used for object allocation. This will make it partially full, and it can be added to the slab cache. This keeps things nice, fast and simple – we maintain far less state.

6.5 Summary and Further Reading

6.5.1 Summary

Summary 6.5.1

1. Managing the available memory is a very important problem in operating systems design.

2. In the old days when there was no virtual memory, the base-limit scheme was used. The CPU-generated address is added to the contents of the base register. If it is less than the contents of the limit register, then it is a valid address. This translated address is sent to the memory system.

3. The stack distance is normally used to quantify temporal locality. All the accessed pages are organized as a stack. Whenever there is a fresh access, the distance of the page in the stack from the top of the stack is the *stack distance*. It is subsequently removed from its current location and pushed to the stack (becomes the new stack top). The distribution of the stack distance is typically a heavy-tailed distribution.

4. The *stack property* is followed by all stack-based page replacement algorithms. Stack property: For the same access sequence and replacement scheme, a larger memory always includes all the pages that are contained in a smaller memory, at all points of time. This ensures that the page fault rate either stays the same or increases monotonically as we increase the size of the memory.

5. The cost function of a page determines its replacement priority. In the case of the optimal algorithm (OPT), it is equal to the time at which the page will subsequently be accessed. This is not known in a practical setting. It can be proven with an exchange-based argument that such an algorithm is optimal in terms of minimizing the page fault rate.

6. The LRU (least-recently used) algorithm uses the reciprocal of the last-accessed time as the cost function. Both LRU and OPT follow the stack property.

7. The FIFO algorithm uses the following cost function: it is the reciprocal of the time at which a page was added to the memory the last time. This is dependent on the size of the memory. It is not memory-dependent. Due to such a cost function, it does not follow the stack property. This leads to the Belady's anomaly. There are access sequences for which a smaller memory can have a significantly lower page fault rate than a larger memory.

8. It is hard to implement LRU in practice. Hence, access tracking is typically done by inducing soft page faults. The WS-Clock and Second Chance algorithms use such a mechanism. If a page has been recently accessed, it is marked as not-accessed and not considered for eviction.

9. The working set is the set of pages that a process accesses in a short duration of time. If the working set does not fit in memory, the page fault rate will be high. It will be much lower if the working set fits in memory.

10. The kernel uses a 48 or 57-bit virtual address. Half of this is reserved for user processes. The rest half is reserved for the kernel. 32 PB is a direct-mapped region where the physical address is a linear function of the virtual address.

11. Linux uses a 5-level page table.

12. Each page table entry stores permission bits.

13. Linux stores all page-related metadata in a `struct page`. It has generalized the notion of pages to folios. A folio contains N pages where N is a power of 2. It is contiguously allocated in the virtual and physical address spaces.

14. Flushing the TLB has serious performance implications because page walks are expensive. On Intel x86 machines, the notion of PCIDs is supported. It can be mapped to a *pid* in Linux. This allows the kernel to selectively flush pages belonging to a certain process. There is no requirement to flush the TLB if the executing userspace process changes.

15. The physical memory space is divided into zones. Each zone indicates the type of access. Some zones are meant to be accessed by the DMA controller, some are movable pages and some are allocated to nonvolatile devices.

16. Reverse mapping is an important element of the virtual memory subsystem. It maps a physical page to all the virtual pages that map to it.

17. The task is to map a physical page to all the `vmas` that have a page that maps to it. This is achieved by creating the following structure.

   (a) Every physical page points to an `anon_vma`, which is just a stub.

   (b) For a given virtual memory region, each `vma` is associated with an `anon_vma` and an `anon_vma_chain` (`avc`), which connects both of them.

   (c) Because a task can get forked, it is possible for multiple `vmas` across processes to be associated with the same `anon_vma` (of the parent process). Each such connection requires a dedicated `avc`.

   (d) Due to a copy-on-write, there can be multiple `anon_vmas` associated with the same `vma`. Again each connection is established through a dedicated `avc`.

   (e) All the `avcs` of a given task are connected to each other using a doubly-linked list.

   (f) All the `avcs` that link all the `vmas` associated with a single `anon_vma` are organized as a red-black tree.

   (g) Using this red-black tree, it is possible to access all the `vmas`. physical page → `anon_vma` → all the `vmas`

   (h) Every physical page maintains its index in the `vma` in which it was originally allocated. This index is used to map a physical page to a virtual page in a `vma` in $\theta(1)$ time.

18. The MGLRU algorithm divides pages into generations. Every generation has a sequence number. The earliest generation has the lowest sequence generation and the newest generation has the highest sequence number.

   (a) Automatically incrementing the current generation *ages* all the pages.

   (b) There are complex rules for deciding when to increment the sequence number. The general idea is to try to balance the number of pages across generations.

   (c) The kernel tries to shrink zones by evicting and reclaiming pages both proactively (using `kswapd`) and on-demand.

   (d) The pages in the earliest generation are targeted for eviction and reclamation. Deciding between whether to evict a *file* (file-backed) or *anon* page (heap, stack, etc.) is a difficult decision. Various factors need to be taken into account such as the `swappiness` variable that indicates the preference and the normalized refault rate.

19. Thrashing refers to a phenomenon where the working set does not fit in memory; this leads to a high page fault rate; there is low CPU activity; the scheduler brings in more processes, which further increases the page fault rate. Ultimately, either processes are killed or the system crashes.

20. The buddy allocator is the default kernel memory allocator. It organizes a contiguous physical address space as a tree of regions, where the number of pages in each region is a power of 2. The leaves of the tree are either allocated to kernel threads or are empty. If there are two unallocated sibling leaves, then they are coalesced and removed. Their parent becomes a leaf with twice the memory.

21. For fixed-size objects, the Slab and Slub allocators are used. They divide memory into fixed-size slabs. This makes it easy to allocate objects and manage the memory space. Every slab has a bit vector, where the ith bit indicates that the ith object in the slab is allocated. The Slab allocator maintains a pool of objects, which has a list of full, partially-full and empty slabs. The Slub allocator is more efficient. It maintains only a list of partially-full slabs. It heavily relies on pointer arithmetic.

6.5.2 Further Reading

Exercises

Ex. 1 — Why do FIFO systems suffer from the Belady's anomaly?

Ex. 2 — Is implementing theoretical LRU practical? Justify your answer.

Ex. 3 — How do we practically implement LRU?

Ex. 4 — Let us say that we want to switch between user-mode processes without flushing the TLB or splitting the virtual address space among user processes. How can we achieve this with minimal hardware support?

* **Ex. 5** — We often transfer data between user programs and the kernel. For example, if we want to write to a device, we first store our data in a character array, and transfer a pointer to the array to the kernel. In a simple implementation, the kernel first copies data from the user space to the kernel space, and then proceeds to write the data to the device. Instead of having two copies of the same data, can we have a single copy? This will lead to a more high-performance implementation. How do we do it, without compromising on security?

Now, consider the reverse problem, where we need to read a device. Here also, the kernel first reads data from the device, and then transfers data to the user's memory space. How do we optimize this, and manage with only a single copy of data?

Ex. 6 — Prove the optimality of the optimal page replacement algorithm.

Ex. 7 — Show that the optimal page replacement algorithm is based on a stack-like property. Will it be susceptible to the Belady's anomaly then? For the second part, use the answer to the first part of the question.

Ex. 8 — Consider a computer system with a 64-bit logical address and an 8-KB page size. The system supports up to 1 GB of physical memory. How many entries are there in a single-level page table and an inverted page table?

Ex. 9 — When and how is the MRU page replacement policy better than the LRU page replacement policy?

Ex. 10 — What is the reason for setting the page size to 4 KB? What happens if the page size is higher or lower? List the pros and cons.

Ex. 11 — Consider a memory that can hold only 3 frames. We have a choice of two page-replacement algorithms: LRU and LFU.

a)Show a page access sequence where LRU is better than LFU?

b)Show a page access sequence where LFU is better than LRU?

Explain the insights as well.

Ex. 12 — What are the pros and cons of prefetching pages?

Ex. 13 — What are the causes of thrashing? How can we prevent it?

Ex. 14 — What is the page walking process used for in the MG-LRU algorithm? Answer in the context of the `lru_gen_mm_state` structure.

Ex. 15 — How is a Bloom filter used to reduce the overhead of page walking?

Ex. 16 — What is the need to deliberately mark actively used pages as "non-accessible"?

Ex. 17 — What is the `swappiness` variable used for, and how is it normally interpreted? When would you prefer evicting FILE pages as opposed to ANON pages, and vice versa? Explain with use cases.

Ex. 18 — Let us say that you want to "page" the page table. In general, the page table is stored in memory, and it is not removed or swapped out – it is basically pinned to memory at a pre-specified set of addresses. However, now let us assume that we are using a lot of storage space to store page tables, and we would like to page the page tables such that parts of them, that are not being used very frequently, can be swapped out. Use concepts from folios, extents and inodes to create such a swappable page table.

Ex. 19 — How is reverse mapping done for ANON and FILE pages?

Ex. 20 — How many `anon_vma` structures is an `anon_vma_chain` connected to?

Ex. 21 — Why do we need separate `anon_vma_chain` structures for shared COW pages and private pages?

Ex. 22 — Given a page, what is the algorithm for finding the *pfn* number of its buddy page, and the *pfn* number of its parent?

Ex. 23 — What are the possible advantages of handing over full slabs to the baseline memory allocation system in the SLUB allocator?

Ex. 24 — Compare the pros and cons of all the kernel-level memory allocators.

Chapter 7

The I/O System, Storage Devices and Device Drivers

There are three key functions of an OS: process manager, memory manager and device manager. We have already discussed the role of the OS for the former two functionalities in earlier chapters. Let us now come to the role of the OS in managing devices, especially storage devices. We have seen in Chapter 1 that most of the kernel code is actually driver code. As a matter of fact, most low-level programmers who work with the kernel are device driver writers. They write drivers for new devices and issue bug fixes for older drivers. Core kernel developers who work on the scheduler or memory system are in comparison much fewer. These are relatively stable parts of the kernel code that don't see that many changes. However, a large number of devices are supported by modern versions of Linux, and each device needs its own custom driver. With the advent of USB technology, there is some degree of consolidation. Now, it is possible for a single generic USB driver to handle multiple devices. For example, a generic keyboard driver can support a large number of USB-based keyboards. Hence, it makes sense to write one complex USB driver that is generic enough to support a large number of devices. This omits the need to write bespoke device-specific drivers. In spite of such competing trends, the area of device drivers is still quite rich and complex. Hence, there is a rich ecosystem comprising driver developers, generic driver codes, software for verifying drivers and code templates.

In the space of devices, storage devices such as hard disks and flash/NVM drives have a very special place. They are arguably the most important citizens in the device world. Other devices such as keyboards, mice and web cameras are nonetheless important, but they are clearly not in the same league as storage devices. The reasons are simple. Storage devices are often needed for a computing system to function. They store all the data required for the system to boot and run important software. A storage device also stores the swap space, which is a key component of the overall virtual memory system.

Linux distinguishes between two kinds of devices: block and character. Block devices read and write a large block of data at a time. For example, storage devices are block devices that often read and write 512-byte chunks of data in one go. On the other hand, character devices read and write a single character or

a set of few characters at a time. Examples of character devices are keyboards and mice. For interfacing with character devices, a simple device driver is often sufficient; it can be connected to the terminal or the window manager. The driver provides the user a method to interact with the underlying OS and applications.

We shall see that managing block devices is more complicated. There is a need to create a custom library that merges, splits and rearranges requests for efficiency. Next, there is a need to create a tree-structured file system to manage all the data stored on the storage device. It cannot be accessed like a RAM that uses virtual memory and a memory map. In any such file system, the internal nodes of the tree are directories (folders in Windows). The leaf nodes are the individual files, also known as regular files. A regular file is defined as a set of bytes that has a specific structure based on the type of data it contains. For instance, we can have image files, audio files, document files, etc. A directory or a folder on the other hand has a fixed tabular structure that just stores pointers to every constituent file or directory inside it.

Linux defines the concept of a generic *file*, which subsumes directories, regular files and a bunch of other entities such as devices, symbolic links, network sockets and virtual files that hold kernel information. We shall look at many such types of files later in this chapter. The basic idea is that Linux allows us to interact with diverse types of entities using a common set of functions. Given that each such entity is thought to be a file, standard file access functions can be used.

Organization of this Chapter

Figure 7.1 shows the organization of this chapter.

The first section explains the basics of the I/O system. It is important to understand the hardware architecture before discussing the software support provided by the kernel. We shall briefly cover the design of the motherboard, the layers in the I/O system and the different forms of I/O on modern systems. We shall briefly touch upon memory-mapped I/O, port-mapped I/O and DMA.

Next, we shall cover the basics of storage devices. They are a special class of I/O devices that store data permanently and thus require extensive support. They store the code and data of all the applications and the OS kernel. Hence, to efficiently access them, it is important to understand their design and limitations. We shall cover hard disks, RAID configurations (redundant array), SSDs and novel NVM devices. We shall later on observe that it is important to know the idiosyncrasies of the underlying storage hardware to design efficient file systems.

We shall then look at the concept of the "generic file" in Linux. Linux represents everything including regular files, directories, devices, kernel interfaces, etc., as a file. In terms of devices, there are two kinds of devices: block devices such as hard disks and character devices such as mice and keyboards. Block devices require elaborate kernel support. They have specialized drivers that have bespoke code for device regstration, block I/O, prefetching and batching, scheduling and monitoring the health of the attached device. Character devices, on the other hand, are not that difficult to manage. Their device drivers are often external modules that are much simpler.

Finally, we will discuss file systems that manage a large number of generic
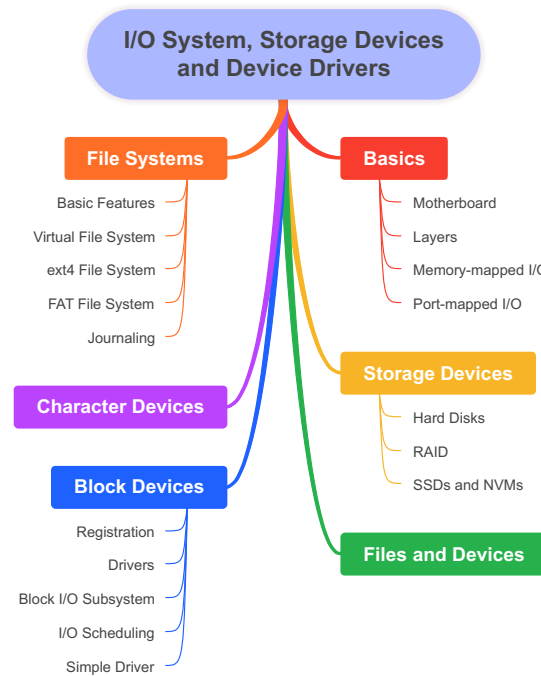
Figure 7.1: Organization of this chapter

files. Linux defines a virtual file system architecture that provides a common interface to for every constituent file system. The main role of a file system is to manage all its constituent files and to also specify the format of each file. For instance, the popular ext4 file system in Linux uses a specific file format to map logical file addresses to 512-byte blocks on the disk. The FAT file system uses a different format. It is popularly used in USB memory sticks these days. Highly reliable file systems use journaling that ensure that all the updates are correctly sent to the storage device and the system is immune to events like power failures. The chapter will end with a discussion of such systems.

7.1 Basics of the I/O System

7.1.1 The Motherboard and Chipset

Figure 7.2 shows a conceptual diagram of the I/O system. On different machines it can vary slightly, however the basic structure is still captured by the figure.

The CPU chip referred to as the processor in the figure is connected to the North Bridge chip using a high-bandwidth bus. This chip connects to the memory chips and the GPU (graphics processor). It has a built-in memory controller that controls all the memory modules and schedules all the memory accesses. The other connection is to a PCI Express link that has many high
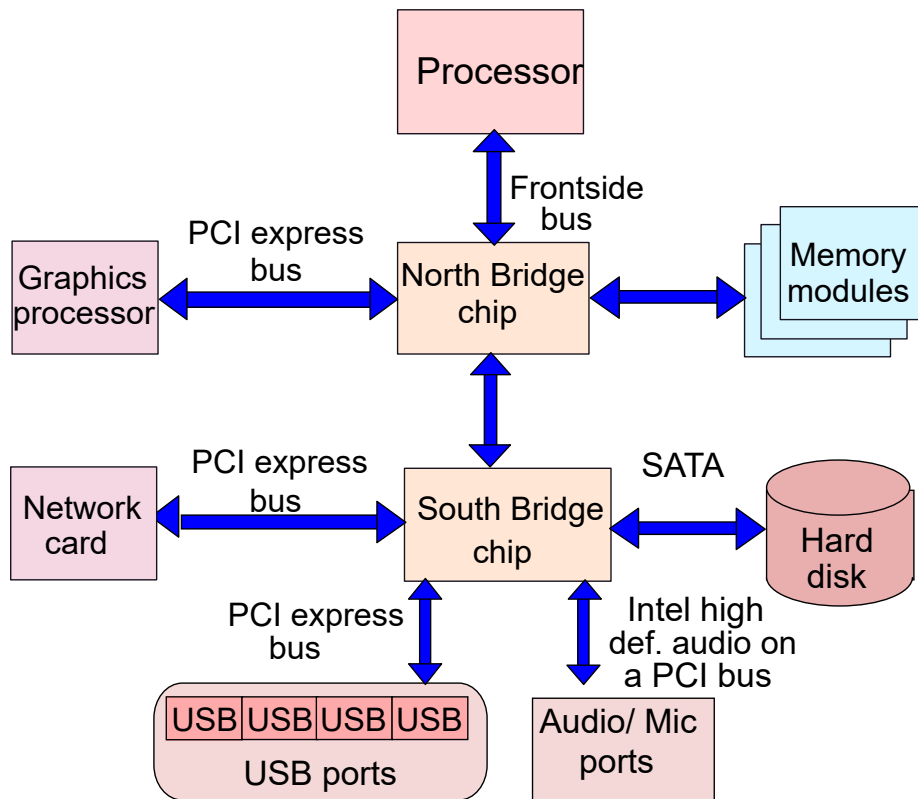
Figure 7.2: The I/O system in classical systems

bandwidth devices on it such as a set of graphics processors. Nowadays, the North bridge has been replaced by on-chip memory controllers. That have dedicated connections to the memory modules via 64 or 72-bit channels. A graphics processor has also moved into the chip. It is known as the integrated graphics card. There is a separate bus that connects all the cores to the on-chip GPU.

The basic idea is that there are dedicated hardware modules that connect the cores and caches to on or off-chip memory modules and GPU hardware. They are typically managed at the hardware level and there is very little OS involvement. Given that these devices have very high bandwidths and very low latencies, no kernel routine can be involved in data transfers. The performance overheads will become prohibitive. They can however be a part of the control path – configure these devices and handle errors.

The job of orchestrating and coordinating regular I/O operations is delegated to the South Bridge chip. It is typically a specialized piece of hardware that is resident on the motherboard. Its job is to interface with a diverse set of I/O devices via their bus controllers. Typically, we connect a multitude of I/O devices to a single set of copper wires known as a *bus*. Each bus has a dedicated bus controller that acts as an arbiter and schedules the accesses of the I/O devices. Some examples are PCI and PCI Express buses that connect to a set

of peripherals like the network cards and USB ports, the audio bus that connects to the speaker output and mic input, and the SATA bus that connect to SATA disk drives. The role of the South bridge chip is quite important in the sense that it needs to interface with numerous controllers corresponding to a diverse set of buses. Note that we need additional chips corresponding to each kind of bus, which is why when we look at the picture of a motherboard, we see many chips. Each chip is customized for a given bus (set of devices). These chips are together known as the *chipset*. They are a basic necessity in a large system with a lot of peripherals. Without a chipset, we will not be able to connect to external devices, notably I/O and storage devices. Over the last decade, the North Bridge functionality has moved on-chip. Connections to the GPU and the memory modules are also more direct in the sense that they are directly connected to the CPUs via either dedicated buses or memory controllers.

However, the South Bridge functionality has remained as an off-chip entity in many general purpose processors on server-class machines. It is nowadays (as of 2024) referred to as the Platform Controller Hub (PCH). Modern motherboards still have a lot of chips including the PCH primarily because there are limits to the functionality that can be added on the CPU chip. Let us elaborate.

I/O controller chips sometimes need to be placed close to the corresponding I/O ports to maintain signal integrity. For example, the PCI-X controller and the network card (on the PCI-X bus) are in close proximity to the Ethernet port. The same is the case for USB devices and audio inputs/ outputs. The other issue is that to connect to a wide variety of peripherals we need a lot of pins on the CPU. We seldom have so many pins available on the CPU package to connect to I/O devices. CPU chips that are designed for larger machines do not have enough free pins – most of their pins are used to carry current (power and ground pins). Hence, we need to multiplex I/O devices on the same set of pins. Connecting all of them to the PCH (South Bridge chip) is a way of achieving this. The PCH multiplexes a wide variety of devices, schedules their requests and efficiently manages the I/O traffic.

In many mobile phones, a lot of the South Bridge functionality has moved to the CPU chip, and thus many I/O controllers are consequently present within the CPU package. This is a reasonable design choice for specific types of mobile devices where we expect to connect to a fixed set of peripherals, the motherboard (which has a small size), and the CPU package has enough pins available because these are low-power devices.

Regardless of which component is inside the chip and which component is outside the chip, the architecture of the I/O subsystem has remained more or less like this (with minor changes) over the years. As discussed in the previous paragraph, depending upon the use case, some components are placed inside the CPU package and some components are placed outside it. There are many factors for making such a decision such as the target use case, the size of the motherboard, the ease of packaging and the target communication latency between the CPUs and the I/O devices.

Figure 7.3 shows the flow of actions when an application interacts with an I/O device. The application makes a request to the kernel via a system call. This request is forwarded to the corresponding device driver, which is the only subsystem in the kernel that can interact with the I/O device. The device driver issues specialized instructions to initiate a connection with the I/O device. A request gets sent to the I/O device via the chips in the chipset. A set of chips
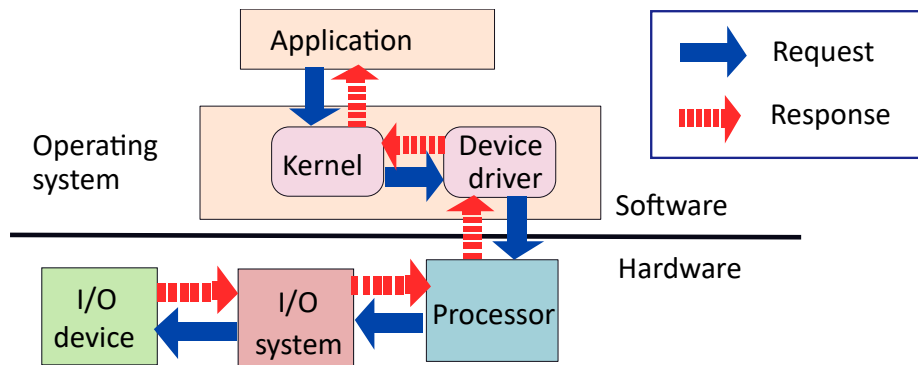
Figure 7.3: Flow of actions in the kernel: application → kernel → device driver → CPU → I/O device (and back)

that are a part of the chip set route the request to the I/O device. The South Bridge chip is one of them. Depending upon the request type, read or write, an appropriate response is sent back. In the case of a read, it is a chunk of data and in the case of a write, it is an acknowledgment.

The response follows the reverse path. Here there are several options. If it was a synchronous request, then the processor waits for the response. Once it is received, the response (or a pointer to it) is put in a register, which is visible to the device driver code. However, given that I/O devices can take a long time to respond, a synchronous mechanism is not always the best. Instead, an asynchronous mechanism is preferred where an interrupt is raised when the response is ready. The CPU that handles the interrupt fetches the data associated with the response from the I/O system.

This response is then sent to the interrupt handler, which forwards it to the device driver. The device driver processes the response. After processing the received data, a part of it can be sent back to the application via other kernel subsystems.

7.1.2 Layers in the I/O System

A modern I/O system is quite complex primarily because we need to interface with many heterogeneous devices. Therefore, there is a need to break down the I/O system into a bunch of layers where the connotation of a layer is the same as that of a layer in the classical 7-layer OSI model for computer networks. The idea of a layer is that it has a foxed functionality in terms of the input that it receives from its lower layer and the outputs that it provides to its upper layer. Furthermore, there is a well-defined interface between a layer and its adjacent layers. This basically means that we can happily change the implementation of a layer as long as it continues to perform the same function and its interfaces with the adjoining layers remain the same – the correctness of the system is not affected. This has allowed computer networks to scale across many heterogeneous devices and technologies. The key idea here is that the layers are independent of each other and thus one implementation of a layer can be easily replaced with another one. For example, the TCP/IP protocol

works for Ethernet-based wired networks, WiFi networks and even 4G and 5G networks. The protocol is independent of the technology that is actually used. This is only possible because of the independence of layers. Similarly, the HTTP protocol is used for accessing websites. It is also layer independent. Hence, it works on all kinds of networks. Figure 7.4 tries to achieve something similar for I/O systems by proposing a 4-layer protocol stack.
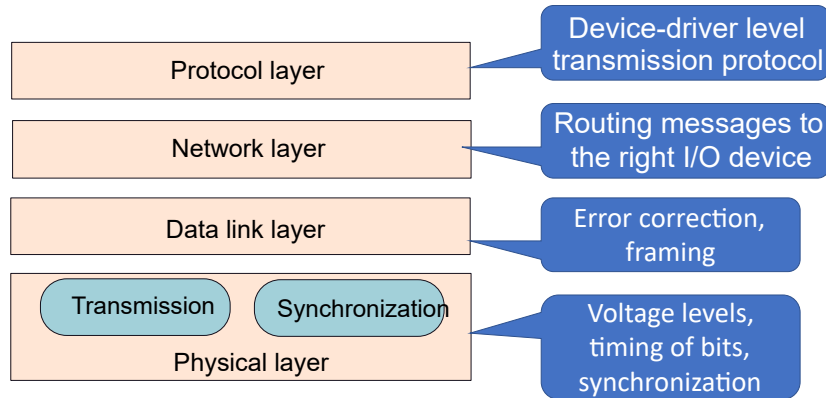
Figure 7.4: 4-layer I/O protocol stack

The lowest layer is the physical layer that is divided into two sublayers namely the transmission sublayer and the synchronization sublayer. The transmission sublayer defines the electrical specifications of the bus, and the way that data is encoded. For example, the data encoding could be active high, which means that a high voltage indicates a logical 1 and vice versa. Or the encoding could be active low, which means that a low voltage indicates a logical 1 and vice versa. In this space, there are many encoding schemes, and they have different properties. The synchronization sublayer deals with timing. We need to understand that if we are sending data at a high frequency, then recovering the data requires a strict clock synchronization between the sender and receiver. There are several options here. The first is that the sender and receiver are proximate, and their clocks are indeed synchronized. In this case, it is easy for the receiver to recover the data. The second option is called a *source synchronous scheme* where the clock is sent along with the data. The data can be recovered at the receiver using the clock that is sent along with the data. In some other cases like USB, the clock can be recovered from transitions in the data itself. In some cases when a string of 0s and 1s are sent, artificial transitions are inserted into the data for easier clock recovery.

The data link layer has a similar functionality as the corresponding layer in networks. It performs the key tasks of error correction and framing (chunk data into fixed sets of bytes).

The network layer routes messages/ requests to a specific I/O device or to the CPU. Every entity placed on the motherboard has a unique address that is assigned to it based on its location. The chips in the chipset know how to route the message to the target device. We have seen in Chapter2 that there are two methods of addressing: based on I/O ports and memory-mapped addressing. This layer converts memory addresses to I/O addresses. The I/O

addresses are interpreted by the chipset and then the I/O requests are sent to the corresponding target devices.

Finally, the protocol layer is concerned with the high-level data transfer protocol. There are many methods of transferring data such as interrupts, polling and DMA. Interrupts are a convenient mechanism. Whenever there is any new data at an I/O device, it simply raises an interrupt. Interrupt processing has its overheads.

On the other hand polling can be used where a thread continuously *polls* (reads the value) an I/O register to see if new data has arrived. If there is new data, then the I/O register stores a logical 1. The reading thread can reset this value and read the corresponding data from the I/O device. Polling is a good idea if there is frequent data transfer. We do not have to pay the overhead of interrupts. We are always guaranteed to read or write some data. On the other hand, the interrupt-based mechanism is useful when data transfer is infrequent. The last method is outsourcing the entire process to a DMA (Direct Memory Access) controller. It performs the full I/O access (read or write) on its own and raises an interrupt when the overall operation has completed. This is useful for reading/ writing large chunks of data.
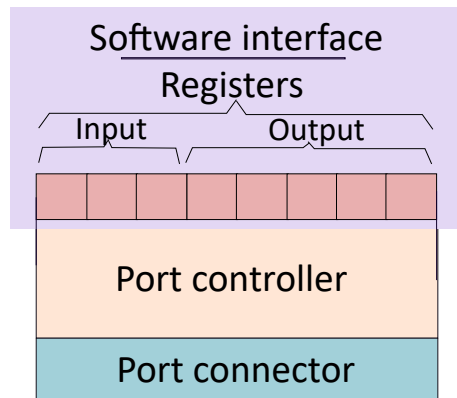
Figure 7.5: I/O ports

7.1.3   Port-Mapped I/O

Let us now understand how I/O devices are accessed. Each device on the motherboard exposes a set of I/O ports. An I/O port in this case is a combined hardware-software entity. A set of I/O ports is a set of registers that are accessible to privileged instructions (refer to Figure 7.5). These registers are of three types: input (read by the CPU), output (written to by the CPU), and read-write registers that can be read or written. The registers are connected to a specialized hardware device called a *port controller*. It interacts with the device through the *port connector*, which is a physical connector. Examples of port connectors include the Ethernet and USB ports. External devices are connected to these via cables or sometimes even directly like USB keys. Each such connector has an associated controller chip that handles the physical and

data link layers. The controller speaks a "low-level language" (language of bits and voltages).

Whenever some data is read, it is placed in the input registers. Similarly, when some bytes need to be sent to the device, the CPU writes them to the output registers. The controller picks the data from these registers, initiates a connection with the I/O device and sends the data. These registers thus act as a hardware front-end for the I/O device. They are accessible to privileged assembly code as software-visible I/O ports. All that we need to do is use x86's *in* and *out* instructions to access these registers.

Intel x86 machines typically define 8-bit I/O ports. The architecture supports 64k (2^{16}) I/O ports. These ports are assigned to the connected devices. A 32-bit register can be realized by fusing four consecutive 8-bit I/O ports. Using this technique, it is possible to define 8-bit, 16-bit and 32-bit I/O ports. An I/O port thus presents itself as a regular register to privileged assembly code, which can be read or written to.

The set of I/O ports form the I/O address space of the processor. The chipset maintains the mapping between I/O ports and the device controllers – this is a part of the network sublayer. In x86, the *in* and *out* instructions are used to read and write to the I/O ports, respectively. Refer to Table 7.1.

Instruction	Semantics
in r1, $\langle i/oport \rangle$	r1 ← contents of $\langle i/oport \rangle$
out r1, $\langle i/oport \rangle$	contents of $\langle i/oport \rangle$ ← r1

Table 7.1: The `in` and `out` I/O instructions in x86

Let us now consider the quintessential method of accessing I/O devices – port-mapped I/O. An I/O request contains the address of the I/O port. We can use the `in` and `out` instructions to read the contents of an I/O port or write to it, respectively. The pipeline of a processor sends an I/O request to the North Bridge chip, which in turn forwards it to the South bridge chip. The latter forwards the request to the destination – the target I/O device. This uses the routing resources available in the chipset. This pretty much works like a conventional network. Every chip in the chipset maintains a small routing table; it knows how to forward the request given a target I/O devices. The response follows the reverse path, which is towards the CPU.

This is a simple mechanism that has its share of problems. The first is that it has very high overheads. An I/O port is 8 to 32 bits wide, which means that we can only read or write 1 to 4 bytes of data at a time. This basically means that if we want to access a high-bandwidth device such as a scanner or a printer, a lot of I/O instructions need to be issued. This puts a lot of load on the CPU's pipeline and prevents the system from doing any other useful work. We need to also note that such I/O instructions are expensive instructions in the sense that they need to be executed sequentially. They have built-in fences (memory barriers). They do not allow reordering. I/O instructions permanently change the system state and thus no other instruction – I/O or regular memory read/write – instruction can be reordered with respect to it.

Along with bandwidth limitations and performance overheads, using such instructions makes the code less portable across architectures. Even if the code is

migrated to another machine, it is not guaranteed to work because the addresses of the I/O ports assigned to a given device may vary. The assignment of I/O port numbers to devices is a complicated process. For devices that are integrated into the motherboard, the port numbers are assigned at the manufacturing time. For other devices that are inserted to expansion slots, PCI-express buses, etc., the assignment is done at boot time by the BIOS. Many modern systems can modify the assignments after booting. This is why, there can be a lot of variance in the port numbers across machines, even of the same type.

Now, if we try to port the code to a different kind of machine, for example, if we try to port the code from an Intel machine to an ARM machine, then pretty much nothing will work. ARM has a very different I/O port architecture. Note that the `in` and `out` assembly instructions are not supported on ARM machines. At the code level, we thus desire an architecture-independent solution for accessing I/O devices. This will allow the kernel or device driver code to be portable to a large extent. The modifications to the code required to port it to a new architecture will be quite limited.

Note that the I/O address space is only 64 KB using this mechanism. Often there is a need for much more space. Imagine we are printing a 100 MB file; we would need a fair amount of buffering capacity on the port controller. This is why many modern port controllers include some amount of on-device memory. It is possible to write to the memory in the port controller directly using conventional instructions or DMA-based mechanisms. GPUs are prominent examples in this space. They have their memory. The CPU can write to it. Many modern devices have started to include such on-device memory. USB 3.0, for example, has about 250 KB of buffer space on its controllers.

7.1.4 Memory Mapped I/O

This is where the role of another technology namely memory mapped I/O becomes very important. It defines a *virtual layer* between the I/O ports and the device driver or user application. In a certain sense, the I/O ports and the device memory (if there is one) *map* to the regular physical address space.

The same TLB and page table-based mechanism are used to map virtual addresses to physical actresses. However, the twist here is that the physical addresses are not stored on a regular DRAM or SRAM-based memory device. Instead, they are actually I/O port addresses or addresses in the internal device memory of a port controller. Given a physical address, the TLB is aware that it is not a regular memory address, it is instead an I/O address. It can then direct the request to the target I/O device. This is easy to do if the physical address space is partitioned. If we do not want to partition the physical address space, then we always have the option of annotating a TLB or page table entry to indicate that the entry corresponds to an I/O address.

We would need specialized hardware support for implementing both these ideas. The summary of the discussion is that the physical address space can be quite heterogeneous, and it can encompass many different kinds of devices, including regular memory as well as various kinds of I/O devices.

It is the role of the operating system, especially the device driver and parts of the kernel, to create a mapping between the I/O ports (or device memory) and physical memory pages. This ensures that the same program that accesses I/O devices can run on multiple machines without significant modifications.

All that it needs to do is issue regular memory load and store instructions. Magically, these instructions get translated to I/O accesses and are sent to the corresponding I/O devices. This makes it very easy to read and write large chunks of data in one go. For example, we can use the `memcpy` function in C for effecting such transfers between DRAM memory and I/O device memory (or registers) very easily. This makes the programming model quite simple.

7.2 Storage Devices

7.2.1 Hard Disks

Let us look into hard disks, the most popular form of storage as of 2024, which perhaps may be phased out in some years from now. Nevertheless, given its towering presence in the area of storage devices, it deserves to be described first. The storage technology is per se quite simple and dates way back to 1957, when IBM shipped the first hard disk. It stores data based on the direction of magnetization of a dipole magnet. There are tiny dipole magnets organized in concentric circles on a disk (a platter), and their direction of magnetization determines the sequence of bits that is stored inside it. Over the last 50 years, it has become the dominant storage technology in all kinds of computing devices starting from laptops to desktops to servers. After 2015, it started to get challenged in a big way by other technologies that rely on nonvolatile memory. However, hard disks are still extremely popular as of 2024, given their scalability and cost advantages.

Storage of Bits on the Recording Surface

To understand how bits are stored on a hard disk, we need to first understand NRZI (non-return to zero inverted) encoding. It is a way of encoding bits in any high-speed I/O device. It is shown in Figure 7.6.
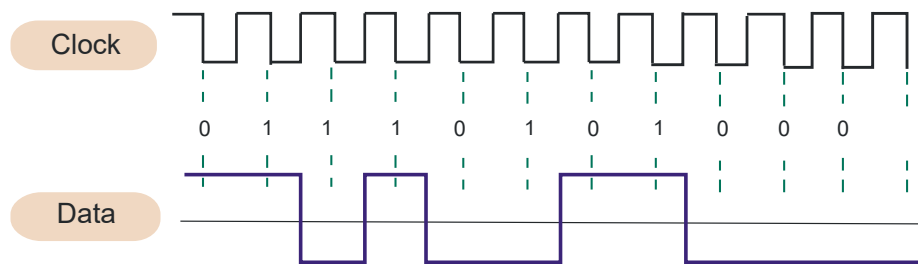
Figure 7.6: NRZI encoding

We need to understand the hard disk read/write head moves very quickly over the recording surface. At periodic intervals, it needs to read the bits stored on the recording surface. Note that the magnetic field is typically not directly measured, instead the change in magnetic field is noted. It is much easier to do so. Given that a changing magnetic field induces a current across terminals on the disk head, this can be detected very easily electronically. Let us say this

happens at the negative edge of the clock. We need perfect synchronization here. This means that whenever the clock has a negative edge, that is exactly when a magnetic field transition should be happening. We can afford to have a very accurate clock but placing magnets, which are physical devices, such accurately on the recording surface is difficult. There will be some variation in the production process. Hence, there is a need to periodically resynchronize the clock with the magnetic field transitions recorded by the head while traversing over the recording surface. Some minor adjustments are continuously required. If there are frequent $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions in the stored data, then such resynchronization can be done.

However, it is possible that the data has a long sequence of 0s and 1s. In this case, it is often necessary to introduce dummy transitions for the purpose of synchronization. In the light of this discussion, let us try to understand the NRZI protocol. A value equal to 0 maintains the voltage value. Whereas, a value equal to 1, flips the voltage. If the voltage is high, it becomes low, and vice versa. A logical 1 thus represents a voltage transition, whereas a logical 0 simply maintains the value of the voltage. It is true that there are transitions in this protocol whenever there is a logical 1, however, if there could still be a long run of 0s. This is where, it is necessary to introduce a few dummy transitions. The dummy data is discarded later.
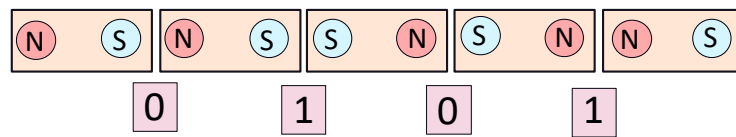
Figure 7.7: Arrangement of tiny magnets on a hard disk's recording surface

Figure 7.7 show the arrangement of magnets on a hard disk's recording surface. As we can see, if the direction of magnetization is the same, then no there is no change in the direction of magnetization. This represents a logical 0. However, whenever there is a logical 1, there is a transition in the direction of the magnetic field. This induces a current, which can be detected. The parallels to NRZI encoding are obvious.

The NRZI encoding is clearly visible in Figure 7.7. If two adjoining magnets have the same direction of magnetization, then there will be no current induced because there is no change in the magnetic field. Recall the Faraday's law where an EMF is induced when a conductor is placed in a time-varying magnetic field. We can thus infer a logical 0. However, if the directions of magnetization are opposite to each other, then there will be a change in the direction of the magnetic field, and this will induce an EMF across the two ends of a conductor (as per the Faraday's law). As per the NRZI encoding, in this case, we can infer a logical 1.

Let us now understand how these small magnets are arranged on a circular disk that is known as a *platter*. As we can see in Figure 7.8, the platter is divided into *concentric rings* that contain such tiny magnets. Each such ring is called a *track*. It is further divided into multiple sectors. Each sector typically has the same size: 512 bytes. In practice, a few more bytes are stored for the sake of error correction. To maximize the storage density, we would like each individual
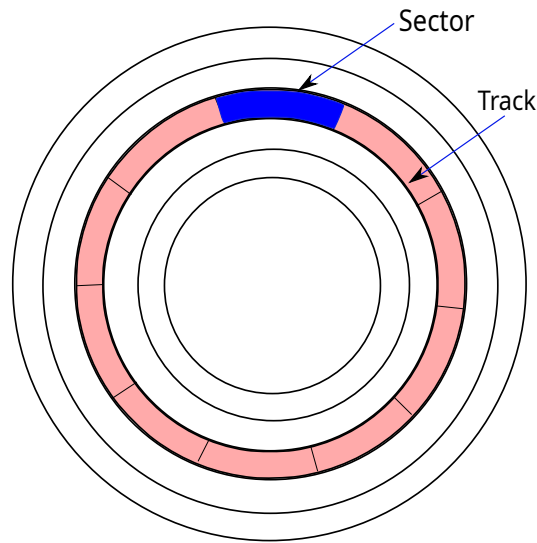
Figure 7.8: Structure of a platter. Note the sectors and tracks.

magnet to be as small as possible. However, there is a trade-off here. If the magnets are very small, then the EMF that will be induced will be very small and will become hard to detect. As a result, there are technological limitations on the storage density.

Hence, it is a wise idea to store different numbers of sectors per track. The number of sectors that we store per track depends on the latter's circumference. The tracks towards the periphery shall have more sectors and track towards the center will have fewer sectors. Modern hard disks are actually slightly smarter. They divide the set of tracks into contiguous sets of rings called *zones*. Each zone has the same number of sectors per track. The advantage of this mechanism is that the electronic circuits get slightly simplified given that the platters rotate at a constant angular velocity. Within a zone, we can assume that the same number of sectors pass below the head each second.

Definition 7.2.1 Key Elements of a Hard Disk

- A disk has a set of platters. A single spindle passes through all of them. Each platter has two recording surfaces.

- Each platter is divided into concentric rings, and each ring is known as a *track*.

- A track stores a set of sectors (512 bytes each, typically).

Design of a Hard Disk

The structure of a hard disk is shown in Figures 7.9 and 7.10. As we can see, there are a set of platters that have a spindle passing through their centers. The spindle itself is controlled by a spindle motor that rotates the platters at a
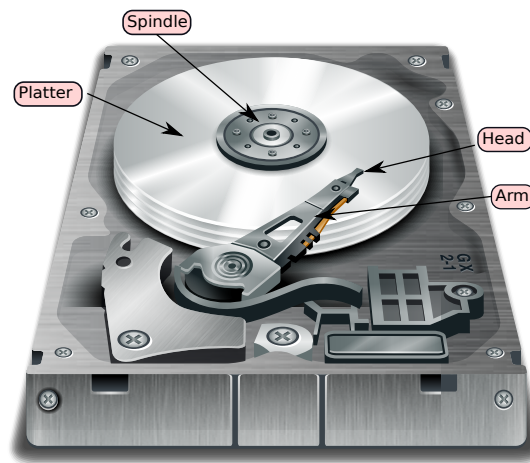
Figure 7.9: A hard disk
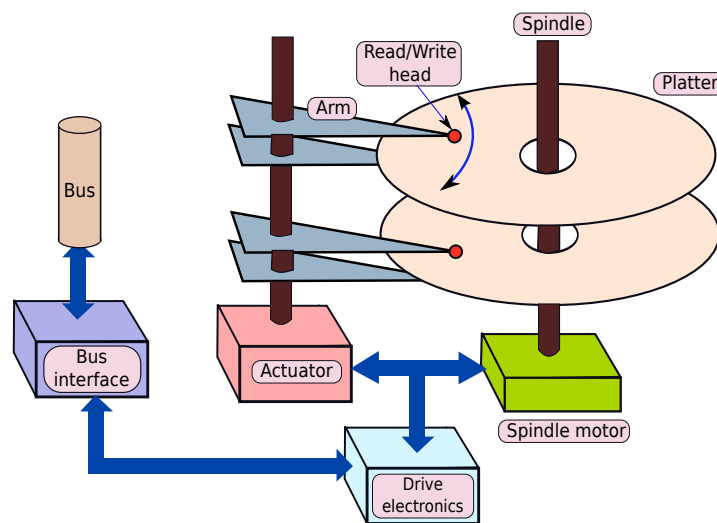
Figure 7.10: Internal structure of a hard disk

constant angular velocity. There are disk heads on top of each off the recording surfaces. These heads are connected to a common *rotating arm*. Each disk head can read as well as write data. Reading data involves sensing whether there is a change in the voltage levels or not (presence or absence of an induced EMF). Writing data involves setting a magnetic field using a small electromagnet. This aligns the magnet on the platter with the externally induced magnetic field. We have a sizable amount of electronics to accurately sense the changes in the magnetic field, perform error correction, and transmit the bytes that are read

back to the processor via a bus.

Let us now understand how a given sector is accessed. Every sector has a physical address. Given the physical address, the disk controller knows the platter on which it is located. A platter can have two recording surfaces: one on the top and one on the bottom. The appropriate head needs to be activated, and it needs to be positioned at the beginning of the corresponding sector and track. This involves first positioning the head on the correct track, which will happen via rotating the *disk arm*. The time required for this is known as the *seek time*. Once the disk head is on the right track, it needs to wait for the sector to come underneath it. Given the fact that the platter rotates at a constant angular velocity, this duration can be computed quite accurately. This duration is known as the rotational latency. Subsequently, the data is read, error checking is done, and after appropriately framing the data, it is sent back to the CPU via a bus. This is known as the transfer latency. The formula for the overall disk access time is shown in Equation 7.1.

$$T_{disk\_access} = T_{seek} + T_{rot\_latency} + T_{transfer} \qquad (7.1)$$

Definition 7.2.2 Disk Access Time Parameters

- The seek time is the time that it takes the head to reach the right track of the disk.

- The rotational latency is the time that the head needs to wait for beginning of the desired sector to come below it after it has been positioned on the right track.

- The transfer time is the time it takes to transfer the sector to the CPU. This time includes the time to perform error checking, framing and sending the data over the bus.

It is important to note that software programs including devices drivers perceive a hard disk or for that matter any storage device as an array of bytes. They access it using a logical address, which is mapped to a physical address by the disk controller. In some cases, this also can be done by the OS if the disk allows access to the raw device. Note that this is rare. Many a time, a small DRAM-backed cache stores the most recent logical to physical mappings as well as some of the data that was recently accessed. This reduces the load on the hard disk.

Given that in a hard disk, there are mechanical parts and also the head needs to physically move, there is a high chance of wear and tear. Hence, disk drives have limited reliability. They mostly tend to have mechanical failures. To provide a degree of failure resilience, the disk can maintain a set of spare sectors. Whenever there is a fault in a sector, which will basically translate to an unrecoverable error, one of the spare sectors can be used to replace this "bad sector".

There are many optimizations possible here. We will discuss many of these when we introduce file systems. The main idea here is to store a file in such a way on a storage device that it can be transferred to memory very quickly. This means that the file system designer has to have some idea of the physical layout of the disk and the way in which physical addresses are assigned to logical

addresses. If some of this logic is known, then the seek time, as well as the rotational latency can be reduced substantially. For instance, in a large file all the data sectors can be placed one after the other on the same track. Then they can be placed in corresponding tracks (same distance from the center) in the rest of the recording surfaces such that the seek time is close to zero. This will ensure that transitioning between recording surfaces will not involve a movement of the head in the radial direction.

All the tracks that are vertically above each other have almost the same distance from the center. We typically refer to a collection of such tracks using the term *cylinder*. The key idea here is that we need to preserve *locality* and thus ensure that all the bytes in a file can quickly be read or written one after the other. Once a cylinder fills up, the head can move to the adjacent cylinder (next concentric track), so on and so forth.

7.2.2 RAID

Hard drives are relatively flimsy and have reliability issues. This is primarily because they rely on mechanical parts, which are subject to wear and tear. They thus tend to fail. As a result, it is difficult to create large storage arrays that comprise hard disks. We need to somehow make large storage arrays resilient to disk failures. There is a need to have some built-in redundancy in the system. The concept of RAID (Redundant Array of Inexpensive Disks) was proposed to solve such problems. Here, the basic idea is to have additional disks that store redundant data. In case a disk fails, other disks can be used to *recover* the data. The secondary objective of RAID-based solutions is to also enhance the bandwidth given that we have many disks that can be used in parallel. If we consider the space of these two dual aims – reliability and performance – we can design many RAID solutions that cater to different kinds of users. The user can choose the best solution based on her requirements.

RAID 0
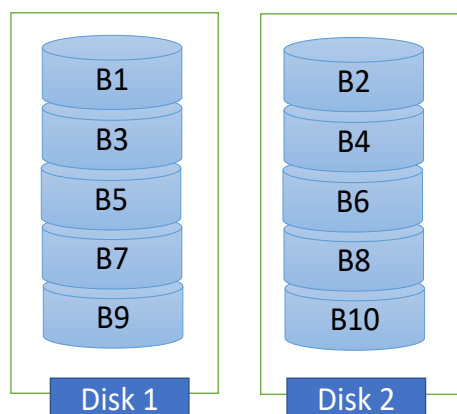
Figure 7.11: RAID 0

Here there is no redundancy, instead a concept called *data striping* is used (refer to Figure 7.11). As we can see in the figure, data is distributed blockwise across the two disks. For example, Disk 1 contains block B1, disk 2 contains block B2, so on and so forth. It is possible to read both the disks in parallel. If we are reading or writing to a large file with a lot of blocks, this strategy effectively doubles the bandwidth. It however does not enhance the reliability.

RAID 1
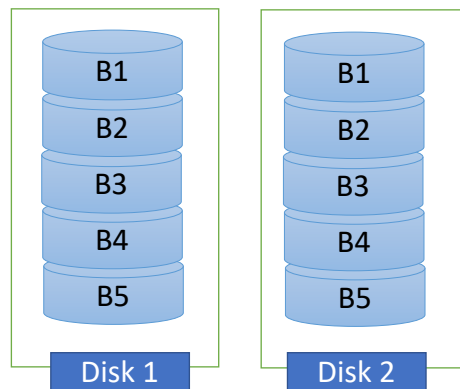
Figure 7.12: RAID 1

On the other hand, RAID 1 (shown in Figure 7.12) enhances the reliability. Here the same block is stored across the two disks. For example, block B1 is stored on both the disks: Disk 1 and 2. If one of the disks fails, then the other disk can be used to service all the reads and writes (without interruption). Later on, if we decide to replace the failed disk then the other disk that is intact can provide all the data to initialize the new disk.

This strategy does indeed enhance the reliability by providing a spare disk. However, the price that is incurred is that for every write operation, we actually need to write the same copy of the block to both the disks. Reads are still fast because we can choose one of the disks for reading. We especially choose the one that is lightly loaded to service the read. This is sadly not possible in the case of write operations.

RAID 2, 3 and 4

We clearly have some issues with RAID 1 because it does not enhance the bandwidth of write operations. In fact, in this case we need to write the same data to multiple disks. Hence, a series of solutions have been proposed to ameliorate this issue. They are named RAID 2, 3 and 4, respectively. All of them belong to the same family of solutions (refer to Figure 7.13).

In the figure we see an array of five disks: four store regular data and one stores parities. Recall that the parity of n bits is just their XOR. If one of the bits is lost, we can use the parity to recover the lost bit. The same can be done at the level of 512-byte blocks as well. If one block is lost due to a disk failure, it can be recovered with the help of the parity block. As we can see in the figure,
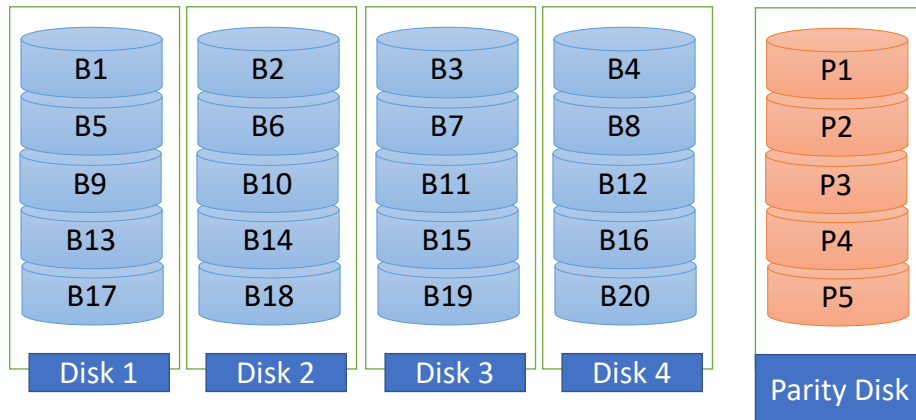
Figure 7.13: RAID 2, 3 and 4

the parity block P1 is equal to $B1 \oplus B2 \oplus B3 \oplus B4$, where $\oplus$ stands for the XOR operation. Assume that the disk with $B2$ fails. We can always compute $B2$ as $B1 \oplus P1 \oplus B3 \oplus B4$.

Let us instead focus on some differences across the RAID levels: 2, 3 and 4. RAID 2 stores data at the level of a single bit. This means that its block size is just a single bit, and all the parities are computed at the bit level. This design offers bit-level parallelism, where we can read different bit streams in parallel and later on fuse them to recreate the data. Such a design is hardly useful, unless we are looking at bit-level storage, which is very rare in practice.

RAID level 3 increases the block size to a single byte. This allows us to read or write to different bytes in parallel. In this case, Disk i stores all the bytes at locations $4n + i$. Given a large file, we can read its constituent bytes in parallel, and then interleave the byte streams to create the file in memory. However, this reconstruction process is bound to be slow and tedious. Hence, this design is also not very efficient nor very widely used.

Finally, let us consider RAID 4, where the block size is equal to a conventional block size (512 bytes). This is typically the size of a sector in a hard disk and thus reconstructing data at the level of blocks is much easier and much more intuitive. Furthermore, it is also possible to read multiple files in parallel given that their blocks are distributed across the disks. Such designs offer a high level of parallelism and if the blocks are smartly distributed across the disks, then a theoretical bandwidth improvement of $4\times$ is possible in this case.

There is sadly a problem with these RAID designs. The issue is that there is a single parity disk. Whenever, we are reading something, we do not have to compute the parity because we assume that if the disk is alive, then the block that is read is correct. Of course, we are relying on block-level error checking, and we are consequently assuming that they are sufficient to attest the correctness of the block's contents. Sadly, in this case writing data is much more onerous. Let us first consider a naive solution.

We may be tempted to argue that to write to any block, it is necessary to read the rest of the blocks from the other disks and compute the new value of the parity. It turns out that there is no need to actually do this; we can instead

rely on an interesting property of the XOR function. Note the following:

$$P1 = B1 \oplus B2 \oplus B3 \oplus B4$$
$$P1' = P1 \oplus B1' \oplus B1 = B1' \oplus B2 \oplus B3 \oplus B4 \tag{7.2}$$

The new parity $P1'$ is thus equal to $B1' \oplus B2 \oplus B3 \oplus B4$. We thus have a neat optimization here; it is not necessary to read the rest of the disks. Nevertheless, there is still a problem. For every write operation, the parity disk has to be read, and it has to be written to. This makes the parity disk a point of contention – it will slow down the system because of requests queuing up. Moreover, it will also see a lot of traffic, and thus it will wear out faster. This will cause many reliability problems, and the parity disk will most likely fail the first. Hence, there is a need to distribute the parity blocks across these disks. This is precisely the problem the novelty of RAID 5.

RAID 5

Figure 7.14 shows a set of disks with distributed parity, where there is no single disk dedicated to exclusively storing parity blocks. We observe that for the first set of blocks, the parity block is in Disc 5. Then for the next set, the parity block $P2$ is stored in Disk 1, so on and so forth. Here the block size is typically equal to the block size of RAID 4, which is normally the disk sector size, i.e., 512 bytes. The advantage here is that there is no single disk that is a point of contention. The design otherwise has the rest of the advantages of RAID 4, which are basically the ability to support parallel read accesses and optimized write accesses. The only disks that one needs to access while writing are as follows: the disk that is being written to and the parity disk.
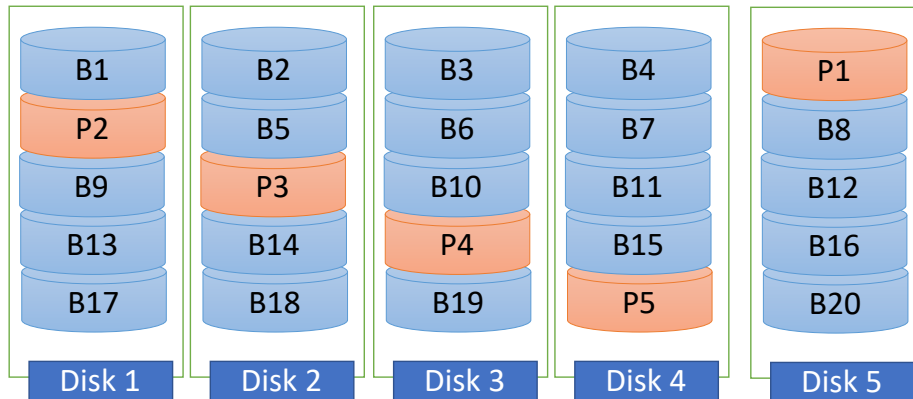
Figure 7.14: RAID 5

RAID 6

Let us now ask a more difficult question, "What if there are two disk failures?" Having a single parity block will not solve the problem. We need at least two
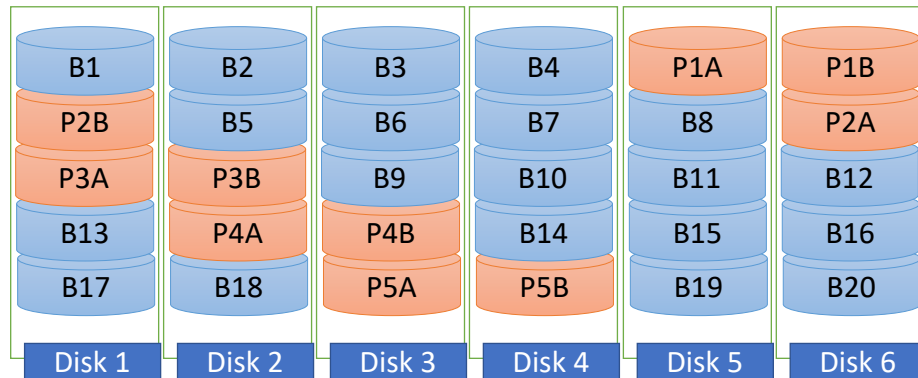
Figure 7.15: RAID 6

parity blocks. The mathematics to recover the contents of the blocks is also much more complex.

Without getting into the intricate mathematical details, it suffices to say that we have two parity blocks for every set of blocks, and these blocks are distributed across all the disks such that there is no point of contention. Figure 7.15 pictorially describes the scheme.

7.2.3 SSDs

Let us next discuss another genre of storage devices that rely on semiconductor technologies. The technology that is used here is known as *flash*. This technology is used to create SSDs (solid state devices). Such storage technologies do not use magnets to store bits, and they also do not have any mechanical parts. Hence, they are both faster and often more reliable as well. Sadly, they have their share of failure mechanisms, and thus they are not as reliable as we think they perhaps are. Nevertheless, we can confidently say that they are immune to mechanical shocks and to a large extent are also immune to fluctuations in temperature.

Basic Operation

Let us understand at a high level how they store a bit. Figure 7.16 shows a novel device that is known as a floating gate transistor. It looks like a normal NMOS transistor with its dedicated source and drain terminals and a gate connected to an external terminal (known as the control gate). Here, the interesting point to note is that there are actually two gates stacked on top of each other. They are separated by an insulating silicon dioxide layer.

Let us focus on the gate that is sandwiched between the control gate and the transistor's channel. It is known as the floating gate. If we apply a very strong positive voltage, then electrons will get sucked into the floating gate because of the strong positive potential and when the potential is removed, many of the electrons will actually stay back. When they stay back in this manner, the cell is said to be *programmed*. We assume that at this point it stores a logical 0. If we wish to *reset* the cell, then there is a need to actually push the electrons back

into the transistor's substrate and clear the floating gate. This will necessitate the application of a strong negative voltage at the control gate terminal, which will push the electrons back into the transistor's body. In this case, the floating gate transistor or the flash cell are said to be *reset*. The cell stores a logical 1 in this state.
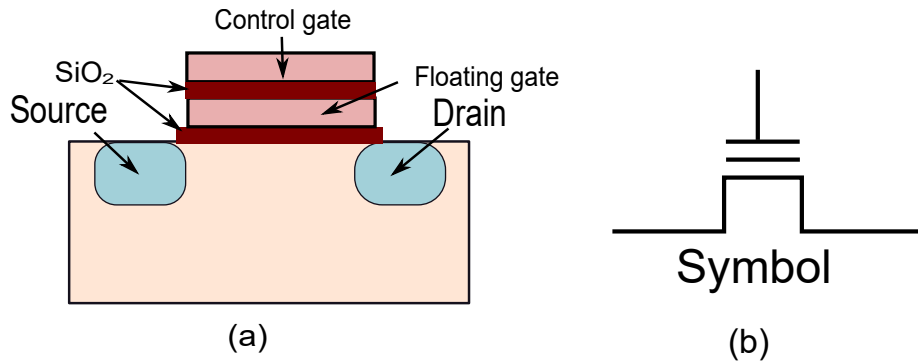
Figure 7.16: Floating gate transistor

Let us now see look at the process of reading the value stored in such a memory cell. When the cell is programmed, its threshold voltage rises. It becomes equal to V_T^+, which is higher than the normal threshold voltage V_T. Hence, to read the value in the cell we set the gate voltage equal to a value that is between V_T and V_T^+. If it is not programmed, then the voltage will be higher than the threshold voltage and the cell will conduct current, otherwise it will be in the cutoff state and will not conduct current. This is known as *enabling* the cell (or the floating gate transistor).

Multi-level Flash Cells

It is possible to increase the storage density even further. We have up till now been considering only two levels: presence of charge in the floating gate or its absence. This automatically translated to two logic levels. Instead, we can have a multilevel flash cell that has 2^n distinct charging levels; this will allow us to store n bits per cell. Such a cell is known as a multilevel flash cell that has a higher storage density. However, it sadly has lower endurance. Additionally, it has a high error rate because small fluctuations in the stored charge can lead to bit errors. This creates a need for more error control bits (ECC bits).

P/E Cycles

Let us now discuss a very fascinating aspect of such flash-based devices. These devices provide read-write access at the level of *pages*, not bytes – we can only read or write a full page (512-4096 bytes) at a time. We cannot access data at a smaller granularity. As we have seen, the storage of data within such devices is reasonably complicated. We have fairly large flash cells and reading them requires some work. Hence, it is a much better idea to read a large number of bytes in one go such that a lot of the overheads can be amortized. Enabling these cells and the associated circuits have associated time overheads, which

necessitates page-level accesses. Hence, reading or writing small chunks of data, let's say a few bytes at a time, is not possible. We would like to emphasize here that even though the term "page" is being used, it is totally different from a page in virtual memory. They just happen to share the same name.

Let us now look at writes. In general, almost all such devices have a DRAM-backed cache that accumulates/coalesces writes. A *write* is propagated to the array of flash cells either periodically, when there is an eviction from the cache, or when the device is being ejected. In all cases effecting a write is difficult mainly because there is no way of directly writing to a flash cell that has already been programmed. We need to first *erase* it or rather deprogram it. In fact, given that we only perform page-level writes, the entire page has to be erased. Recall that this process involves applying a very strong negative voltage to the control gate to push the electrons in the floating gates back into the substrate.

Sadly, in practice, it is far easier to do this at the level of a group of pages, because then we can afford to have a single strong driver circuit to push the electrons back. We can successfully generate a strong enough potential to reset or deprogram the state of a large number of flash cells. In line with this philosophy, in flash-based SSD devices, em blocks are created that contain 32-128 pages. We can erase data only at the level of a block. After that, we can write to a page, which basically would mean programming all the cells that store a logical 0 and leaving/ignoring all the cells that store a logical 1. One may ask a relevant question here, "What happens to the rest of the pages in a block that are not written to?" Let us keep reading to find the answer.

We thus have a program-erase (P/E) cycle. We read or write at the granularity of pages, but erase at the level of blocks (of pages). To rewrite a page, it is necessary to erase it first. This is because the $0 \rightarrow 1$ transition is not possible without an erase operation. The crux of the issue is that we cannot write to a cell that already stores a 0. This means that every page is first written to (programmed), and then erased, then programmed again, so on and so forth. This is known as a program-erase cycle (PE cycle).

Let us understand in detail what happens to the data when we wish to perform a page rewrite operation. Whenever we wish to write to a page, we actually need to do a couple of things. The first is that we need to find another *empty* (not programmed) block. Next, we copy the contents of the current block to the location of the empty block. We omit the page that we wish to write to. This will evolve many read-write operations. Subsequently, we write the modified version of the page. Note that the actual physical location of this page has now changed. Now it is being written to a different location, because the block that it was a part of is going to be erased, and all the other pages that were in its block have already been copied to their new locations in a new block. They are a part of a different physical block now, even though they are a part of the same em logical block. This answers the question with regard to what happens with the rest of the pages in the block.

There is therefore a need to have a table that maps a logical block to its corresponding physical block. This is because, in designs like this, the physical locations of the blocks are changed on every write. Whenever a block is copied to a new address, we update the corresponding mapping. This is done by the Flash Translation Layer (FTL) – typically firmware stored in the SSD itself. The mapping table is also stored on the SSD. It is modified very carefully because we don't want any inconsistencies here. It is seldom the case that the OS maintains

this table. This is because most flash devices do not give access to the OS at this low a level. There are experimental devices known as raw flash devices that allow OS designers to implement the FTL in the OS and subsequently evaluate different mapping algorithms. However, this is a rarity. In practice, even something as simple as a pen drive has its own translation layer.

Reliability Issues

Let us now discuss some reliability issues. Unfortunately, a flash device as of today can only endure a finite number of P/E cycles per physical block. The maximum number of P/E cycle is sadly not much – it is in the range of 50-150k cycles as of 2024. The thin oxide layer breaks down, and the floating gate does not remain usable anymore. Hence, there is a need to ensure that all the blocks wear out almost at the same rate or in other words, they endure the same number of P/E cycles. Any flash device maintains a counter for each block. Whenever there is a P/E cycle, this counter is incremented. The idea is to ensure that all such counts are roughly similar across all the blocks.
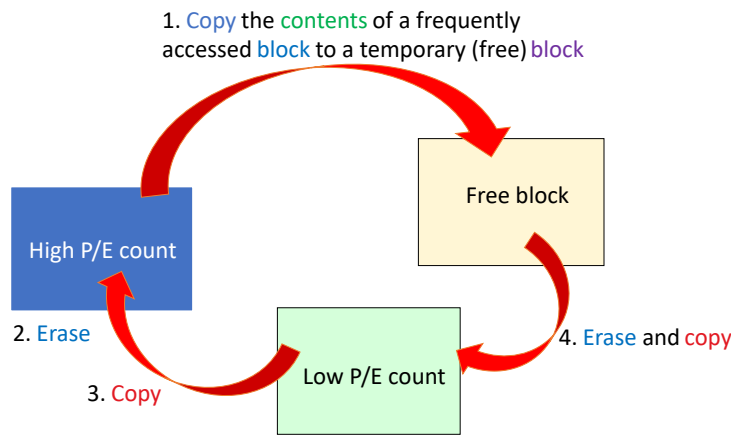
Figure 7.17: Block swap procedure

Sometimes it may so happen that a given block is accessed very frequently and its P/E count increases disproportionately. We need to perform *wear leveling*, which means that we need to ensure that all blocks wear out at roughly the same rate. We thus follow the algorithm shown in Figure 7.17, where we swap its contents with that of a block that has a low P/E count. The steps are as follows.

① We copy the contents of the frequently-accessed (high P/E count) block B to a free/empty block E, whose contents have been erased. ② Next, we erase the contents of block B. ③ In the third step, we transfer the contents of another block C that has a low P/E count to block B. ④ The last step is to copy the contents of block E (erstwhile contents of block B) to block C. This completes the swap. If we have access to some form of volatile storage, such as a DRAM cache, then using the additional free block E that acts as temporary space is not required. The contents of block B can be transferred to DRAM, and then from there to the destination block C that has a low P/E count.

The risk in this process is that the device may be powered off or ejected in the middle of this process. Then all the data that is stored in the volatile DRAM cache can get lost. Hence, this is typically not done. It is way more common to always keep transferring data to a free temporary block such that the system is immune to such kind of events.

Let us now look at another phenomenon that affects reliability. It is known as *read disturbance*. If we read a given transistor continuously in a flash cell, the neighboring transistors are continuously forced to be set to a state where they act like a closed circuit (conduct current). After many such cycles, neighboring transistors may start to get programmed because of the continuous exposure to elevated gate voltages. This is because their threshold voltage is raised to a value that is greater than the threshold voltage corresponding to a logical 0 (programmed state). This means that it is greater than V_T^+.

The increased voltage is not sufficient to program a floating gate transistor. To effect a write, we actually need a much higher voltage. Nevertheless, when we have repeated reads, electrons in neighboring transistors start to accumulate in the respective floating gates gradually. Some electrons shall end up gravitating towards the positive terminal, which in this case is the control gate. This slow accumulation can lead to enough charge accumulating at the floating gate to ultimately program it. Hence, it is important to also maintain a read counter for each block. Whenever it starts to exceed a threshold, we need to copy the block to a new location. This is very similar to the way wear leveling is done.

Performance Considerations

Let us now take a high-level view and go over what we discussed. We introduced a flash cell, which is a piece of nonvolatile memory that retains its values when powered off, quite unlike conventional DRAM memory. Nonvolatile memories essentially play the role of storage devices. They are clearly much faster than hard disks, and they are slower than DRAM memory. However, this does not come for free. There are concomitant performance and reliability problems that require both OS support, and features such as wear leveling and swapping blocks to minimize read disturbance.

Modern SSD devices take care of a lot of this within the confines of the device itself. Nevertheless, operating system support is required, especially when we have systems with large flash arrays. It is necessary to equally distribute requests across the individual SSD memories. This requires novel data layout and partitioning techniques. Furthermore, we wish to minimize the *write amplification*. This is the ratio of the number of physical writes to the number of logical writes. Writing some data to flash memory may involve many P/E cycles and block movements. All of them increase the write amplification. This is why there is a need to minimize all such extraneous writes that are made to the SSD drive.

Most modern SSD disk arrays incorporate many performance optimizations. They do not immediately erase a block that has served as a temporary block and is not required anymore. They simply mark it as invalid, and it is later erased or rather garbage collected. This is done to increase performance. Moreover, the OS can inform the SSD disk that a given block is not going to be used in the future. It can then be marked as invalid and can be erased later. Depending upon its P/E count, it can be either used to store a regular block, or it can even

act as a temporary block that is useful during a block swap operation. The OS also plays a key role in creating snapshots of file systems stored on SSD devices. These snapshots can be used as a backup solution. Later on, if there is a system crash, then a valid image of the system can be recovered from the stored snapshot.

Up till now, we have used SSD drives as storage devices (as hard disk replacements). However, they can be used as regular main memory as well. Of course, they will be much slower. Nevertheless, they can be used for capacity enhancement. There are two configurations in which SSD drives are used: vertical and horizontal. The SSD drive can be used in the horizontal configuration to just increase the size of the usable main memory. The OS needs to place physical pages intelligently across the DRAM and SSD devices to ensure optimal performance. The other configuration is the vertical configuration, where the SSD drive is between the main memory and the hard disk. It acts like a cache for the hard disk – a faster storage device that stores a subset of the data stored on the hard disk. In this case also, the role of the OS is crucial.

7.2.4 Nonvolatile Memories

Akin to flash devices, we can design a host of nonvolatile memories with different kinds of technologies. All of these primarily function as storage devices but can also be used to extend the available physical memory space as well. Many of them have better endurance and shelf lives as compared to flash-based SSD drives.

Their basic underlying philosophy is the same. We construct a device with two physical states: one corresponds to a logical 0 and the other corresponds to a logical 1. Typically, one of them is a high-resistance state and the other is a low-resistance state. Next, we proceed to create an array of such devices. We then enhance its reliability using a combination of hardware and software approaches.

Let us consider a list of such technologies (refer to Table 7.2).

7.3 Files and Devices in Linux

7.3.1 Devices in Linux

Linux defines two kinds of devices: block devices and character devices. A *block device* typically corresponds to storage devices that store data at the granularity of blocks (ten to hundreds of bytes). The minimum data transfer size (reads of writes) is one *block*. Blocks, in general, can be randomly accessed. It is not possible to access data at a finer granularity. A block device driver needs to take all such characteristics of the block device into account. For example, hard disks and SSDs are block devices. We have already seen that a typical block size in a hard disk is 512 bytes, whereas in an SSD we read/write data at the granularity of pages and erase data at the granularity of blocks.

On the other hand, there are *character devices* such as keyboards and mice. They transfer data one or a few bytes at a time. Such devices typically don't have addressable locations and do not function as storage devices. Such devices either read or produce a stream of characters. There could be the notion of

Device	Operating Principle
Flash memory	A floating gate transistor that has an additional floating gate that can be made to store electrons. The presence and absence of electrons in the floating gate corresponds to the two logical states.
Ferroelectric RAM (FeRAM)	The degree of polarization of the ferroelectric material is a function of the voltage applied to it in the past. The current polarization direction represents the logical bit.
Magnetoresistive RAM (MRAM)	The direction of magnetization of a ferromagnetic material is a function of the direction of current flow through it (in the past). The direction represents the logical bit.
Phase change memory (PCM)	We use a small heater to change the state from amorphous to crystalline (two states).
Resistive RAM (ReRAM)	Based on the history of the voltage applied, a filament forms based between the anode and cathode. The resistance and the logical state of the cell is determined by the width of this filament.

Table 7.2: Different kinds of nonvolatile memory (NVM) technologies

a position within a stream; however, that is not integral to the definition of a character device. The device driver for a character device needs to be very different from a block device driver. Data that is meant to be read or written needs to be handled and managed very differently.

Linux follows the "everything is a file" model similar to Unix. This means that every entity in the operating system is treated as a file regardless of whether it is a file or not. For example, devices are treated as files. They are stored in the /dev directory, and can be accessed as regular files. Let us elaborate. The type of the file can be found out by running the command ls -l. An entry is of the form -rw-r--r--. The first character is '-', which means that it is a regular file. It can indicate other types of files as well (refer to Table 7.3).

Indicator	File type
-	regular file
d	directory
l	symbolic link (points to another file or directory)
c	character device
b	block device
s	socket (for inter-process communication)
p	pipe (for inter-process communication)

Table 7.3: Different kinds of files in Linux

7.3.2 Notion of Files

The interesting part is that all such kinds of entities are treated as a "file" by the operating system and support basic file operations: opening and closing a file for access, reading and writing bytes, etc. This is like a virtual layer like a superclass in an object-oriented programming language. Each such high-level operation is then translated to an entity-specific call that implements the operation at a lower level.
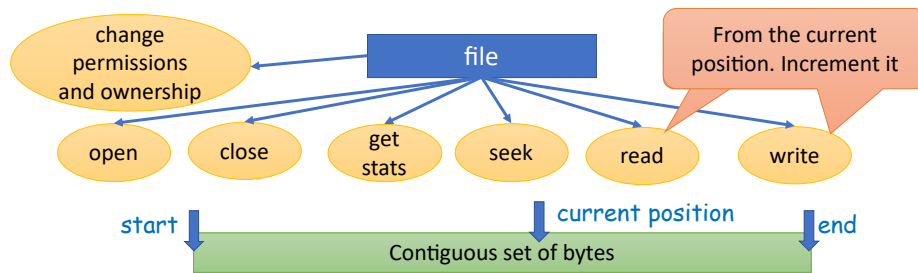
Figure 7.18: File operations in Linux

Every file has a name and associated metadata. The metadata contains its access permissions and ownership information. The owner is a legitimate user who is authorized to access the file. We shall look more at access permissions in Section 8. They basically specify what the owner, other users in the owner's group and the rest of the users can do with the file. Some users have read permission, some can write to it and some can treat it as an executable file. The metadata also contains other statistics (in Linux's terminology) such as the file size, time of last modification, time of last access and the time that the status changed last.

Before accessing a file, it is necessary to *open* it first. This lets the kernel know that the process issuing the system call is accessing the file. Some data structures are initialized to maintain the status of the file access. A file is treated as an array of contiguous bytes. If the process has read 8 bytes, then its current position is set to 8 (assume that we start counting from 0). The *current position* is a part of the bookkeeping information that is maintained for each process. Sometimes a file needs to be locked, especially when multiple processes are concurrently accessing the same file. This information is also maintained in the bookkeeping information maintained along with each open file. The state associated with each open file needs to be cleaned up when the file is closed.

Given that every file is treated as a contiguous set of bytes, where the first byte is located at position 0, it is necessary to maintain the current position (a byte pointer). It is known as a *file pointer*. If we do not maintain the position, then with every *read* or *write* system call, it will be necessary to provide the file pointer (offset within the file). For example, if we would like to read 4 bytes from the file pointer 100 onwards, then the value "100" needs to be provided as an argument to the *read* or *write* system call. This is fine if we have a random file access pattern. However, most files are not accessed in this manner. Typically, they are accessed sequentially. Hence, it is a good idea to maintain an internal file pointer that need not be visible to programmers. From the programmer's

point of view, it is maintained implicitly.

Sequential and Random Access of Files

Sometimes there is a need to access parts of the file randomly. One option is to explicitly manipulate the file pointer. This can be done with the seek family of system calls. The other option is to map parts of the file to memory. This means that file contents are stored in pages that are stored in the regular CPU's memory system. The kernel keeps a record of which page in memory is mapped to which 4 KB chunk in the physical file that is stored on a storage device. The advantage of this mechanism is that all reads and writes can be serviced by the mapped pages. On a regular basis, they can be flushed to the storage device, especially if the memory is starting to filling up. The general idea here is that storage devices are very slow, and we would not like to access them for every file read and write operation. It is a much better idea to create a *page cache* in memory that stores a set of pages that are mapped to contiguous 4 KB chunks of files. All that the program needs to do is write to these pages, which is much faster.

This sounds like a good idea; however, it seems to be hard to implement. How does the program know if a given file offset is present in memory (in a mapped page), or is present in the underlying storage device? Furthermore, the page that a file address is mapped to, might change over the course of time. Thankfully, there are two easy solutions to this problem. Let us consider memory-mapped I/O, where file addresses are directly mapped to virtual memory addresses. In its quintessential form, the TLB is supposed to identify that these are actually I/O addresses, and redirect the request to the I/O (storage) device that stores the file. This is something that we do not want in this case. Instead, we can map the memory-mapped virtual addresses to the physical addresses of pages, which are stored in the page cache. In this case, the target of memory-mapped I/O is a set of another pages located in the memory itself. These are pages that are a part of the page cache. This optimization will not change the programmer's view and programs can run unmodified, albeit much faster. Memory mapping files is of course not a very scalable solution and does not work for large files.

The other solution is when we use I/O-mapped I/O. This means that I/O is performed on files using *read* and *write* system calls, and the respective requests go to the I/O system. They are ultimately routed to the storage device. Of course, the programmer does not work with the low-level details. She simply invokes library calls and specifies the number of bytes that need to be read or written, including their contents (in case of writes). The library calls curate the inputs and make the appropriate system calls. After transferring data to the right kernel buffers, the system call handling code invokes the device driver routines that finally issues the I/O instructions. This is a long and slow process. Modern kernels optimize this process. They hold off on issuing I/O instructions and instead effect the reads and writes on pages in the page cache. This is a much faster process and happens without the knowledge of the executing program.

Let us look now at the data structures used to manage devices in Linux in detail.

7.4 Block Devices
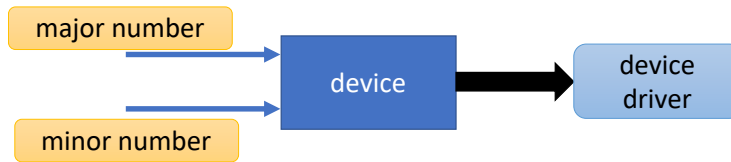
Figure 7.19: Numbering of a device

The first step in incorporating a block device is to register it (refer to Figure 7.19). Every device is assigned a major number and minor number by the kernel. The major number is used to identify the device and the minor number is used internally by the device driver. For example, USB flash drives use the same major number, and thus share the same driver. However, individual devices are assigned different minor numbers.

7.4.1 Registering a Block Device

The first task is to register a block device. The device is registered by invoking the function `register_blkdev` (refer to Listing 7.1). A major number is assigned to it along with a textual name. The most important argument from an operational standpoint is the function pointer `probe` that takes a `devt` (device type) as the sole argument. The device type is a 2-tuple of the 12-bit major number and 20-bit minor number. It is invoked whenever the system boots up and adds the block device, or when it is dynamically inserted. Its job is to initialize the driver, the device and load the code of the device driver in memory.

Listing 7.1: `register_blkdev` function
source : include/linux/blkdev.h

```
register_blkdev (unsigned int major, const char *name,
    void (*probe) (dev_t devt));
```

Let us summarize the current state of our discussion. A major number is assigned to a device or even a family of related devices. If there are many devices of the same type, such as USB devices, the minor number identifies a particular device. The device driver is identified on the basis of the major number. It is possible that a single device driver services multiple devices, which means that it is associated with multiple major numbers. As we shall shortly see, a device driver presents itself to the kernel as a set of function pointers. Given that any modern operating system has to deal with hundreds of devices it needs to incorporate a plethora of device drivers. As a matter of fact, the success and acceptability of an OS is dependent upon its device support. There is thus a need to be very flexible in this regard and support as many devices as possible. Unless OS developers encourage a large community of driver developers to develop drivers for them, users will not adopt the OS because their devices will not work.

7.4.2 Drivers and Modules

Therefore, a standardized interface needs to be provided to driver developers such that they can write drivers easily. Linux indeed makes this process quite simple. A driver's external interface is a set of function pointers. A generic driver is like a pure virtual class in C++ or an abstract class in Java that just defines the signatures of the methods. It is now up to the driver developer to provide concrete implementations of these functions. We have already seen one such function that was sent as an argument to the `register_blkdev` function namely the `probe` function. We shall see that there are many more such functions, which driver developers need to implement.

Before looking at the structure of a driver, let us understand how drivers themselves are manage by the kernel. We have already seen that in current versions of the kernel (as of 2024), 70The reason for this is simple. There are a very wide variety of devices out there starting from printers to web cameras to hard drives to keyboards to mice. Each one of them requires its own driver. For a long time, the adoption of Linux was somewhat subdued primarily because of the limited device support. Over the years, the situation has changed, which is why can we see the disproportionate fraction of driver code in the overall code base. As Linux gets more popular, we will see more driver code entering the codebase. Note that the set of included drivers is not exhaustive. There are still a lot of devices whose drivers are not bundled with the operating system distribution. The drivers have to be downloaded separately. Many times there are licensing issues, and there is also a need to reduce the overall size of the OS install package.

Let us ask an important question at this stage. Given that the Linux kernel is a large monolithic piece of code, should we include the code of all the drivers also in the kernel image? There are many reasons for why this should not be done. The first reason is that the image size will become very large. It may exhaust the available memory space and little memory will be left for applications. The second is that very few drivers may actually be used because it is not the case that a single system will be connected to 200 different types of printers, even though the code of the drivers of these printers needs to be bundled along with the OS code. The reason for this is that when someone is connecting a printer, the expectation is that things will immediately work and all drivers will get auto-loaded.

In general, if it is a common device, there should be no need to go to the web and download the corresponding driver. This would be a very inefficient process. Hence, it is a good idea to bundle the driver along with the OS code. However, bundling the code does not imply that the compiled version of it should be present in the kernel image all the time. Very few devices are connected to a machine at runtime. Only the images of the corresponding drivers should be present in memory.

Modules

Recall that we had a very similar discussion in the context of libraries in Appendix B. We had argued that there is no necessity to include all the library code in a process's image. This is because very few library functions are used in a single execution. Hence, we preferred dynamic loading of libraries and cre-

ated shared objects. It turns out that something very similar can be done here. Instead of statically linking all the drivers, the recommended method is to create a kernel module, which is nothing but a dynamically linked library/shared object in the context of the kernel. All the device drivers should preferably be loaded as modules. At run time they can be loaded on demand. This basically means that the moment a device is connected, we find the driver code corresponding to it. It is loaded to memory on-demand the same way that we load a DLL. The advantages are obvious: efficiency and reduced memory footprint. To load a module, the kernel provides the *insmod* utility that can be invoked by the superuser – one who has administrative access. The kernel can also automatically do this action, especially when a new device is connected. There is a dedicated utility called *modprobe* that is tasked with managing and loading modules (including their dependences).

The role of a module-loading utility is specifically as follows:

1. Locate the compiled code and data of the module, and map its pages to the kernel's memory space.

2. Concomitantly, increase the kernel's runtime code size and memory footprint.

3. There is a need to use a dynamic linker to change the addresses of all the symbols in the module that is going to be added. Similar to the symbol table in regular processes, the kernel maintains a global symbol table. It has a list of all the symbols, functions and variables that are exported by all the modules and the core kernel. These symbols can be used by modules depending upon their requirements. The addresses of all relocatable symbols in the module are updated (akin to symbols in DLLs/shared objects).

4. The symbols exported by the module is added to the global symbol table.
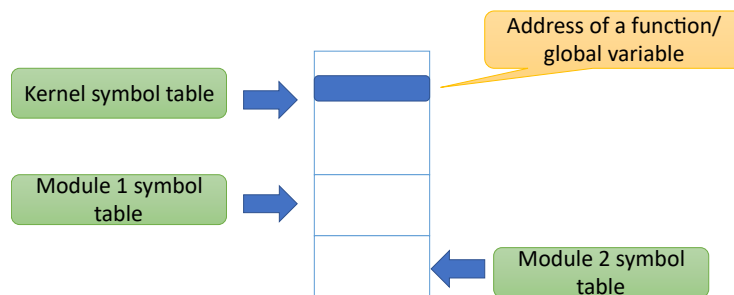
Figure 7.20: Kernel's global symbol table

Global Symbol Table

Figure 7.20 shows the global symbol table. First, we have all the symbols exported by the current kernel, and then we have module-specific symbol tables. They store all the symbols exported by each module. All the modules can use

the symbols exported by the kernel. As we have discussed, they can use symbols exported by other modules as well. Many times there is a need to enforce an order in which the modules are loaded because the nth module may need specific symbols exported by modules 1 to $(n-1)$.

Unloading a module is comparatively much simpler. We maintain a reference count for each module. This maintains a count of the number of other modules including the kernel that use symbols exported by the module. Once the count reaches zero, the kernel can unload the module. We can follow the reverse sequence of steps. We need to clean up the global symbol table, i.e., expunge the symbols exported by the module that needs to be unloaded.
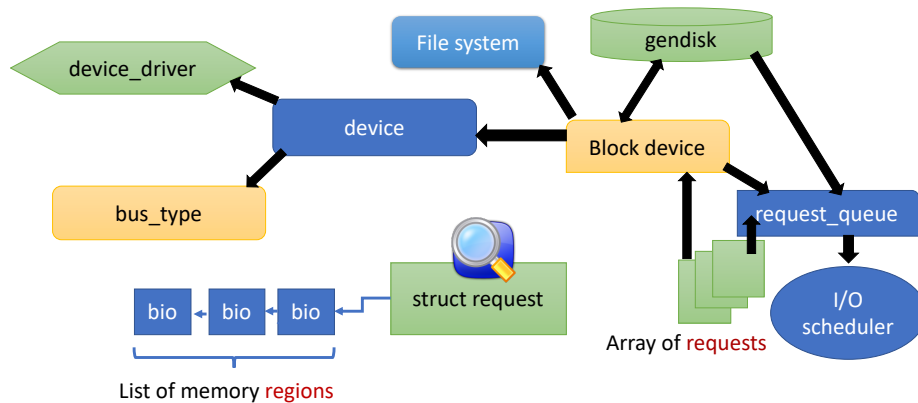
Figure 7.21: Kernel's block I/O system

7.4.3   The Block I/O System

Now we are in a position to appreciate the overall structure of the block I/O subsystem in Linux. It is shown in Figure 7.21. Central to the overall structure of the system are two types of objects namely generic devices (*struct device*) and block devices.

A `device` is a generic construct that can represent both character and block devices. It points to a device driver and a bus. A bus is an abstraction for a shared hardware interconnect that connects many kinds of devices. Examples of such buses are USB buses and PCI Express (PCIe) buses. A bus has associated data structures and in some cases even device drivers. Many times it is necessary to query all the devices connected to the bus and find the device that needs to be serviced. Hence, the generic device has two specific bindings: one with device drivers and one with buses.

Next, let us come to the structure of a block device. It points to many others important subsystems and data structures. First, it points to a file system (discussed in detail in Section 7.6) – a mechanism to manage the full set of files on a storage device. This includes reading and writing the files, managing the metadata and listing them. Given that every block device stores blocks, it is conceptually similar to a hard disk. We associate it with a `gendisk` structure, which represents a generalized disk. We need to appreciate the historical significance of this choice of the word "gendisk". In the good old days, hard disks

were pretty much the only block devices around. However, later on many other kinds of block devices such as SSD drives, scanners and printers came along. Nevertheless, the `gendisk` structure still remained. It is a convenient way of abstracting all of such block devices. Both a block device and the `gendisk` are associated with a request queue. It is an array of requests that is associated with a dedicated I/O scheduler. A `struct request` is a linked list of memory regions that need to be accessed while servicing an I/O request.

Generic Device Driver

Listing 7.2: `struct device_driver`
source : include/linux/device/driver.h

```
struct device_driver {

    /* generic parameters */
    const char *name;
    struct bus_type *bus;
    struct module *owner;

    struct of_device_id *of_match_table;

    /* function pointers */
    int (*probe) (struct device *dev);
    void (*sync_state)(struct device *dev);
    int (*remove) (struct device *dev);
    void (*shutdown) (struct device *dev);
    int (*suspend) (struct device *dev, pm_message_t state);
    int (*resume) (struct device *dev);
}
```

The generic structure of a device driver is shown in Listing 7.2. `struct device_driver` stores the name of the device, the type of the bus that it is connected on and the module that corresponds to the device driver. It is referred to as the `owner`. It is the job of the module to run the code of the device driver.

The key task of the kernel is to match a device with its corresponding driver. Every device driver maintains an identifier of type `of_device_id`. It encompasses a name, a type and other compatibility information. This can be matched with the name and the type of the device.

Next, we have a bunch of function pointers, which are the callback functions. They are called by other systems of the kernel, when there is a change in the state. For example, when a device is inserted, the `probe` function is called. When there is a need to synchronize the state of the device's configuration between the in-memory buffers and the device, the `sync_state` function is called. The `remove`, `shutdown`, `suspend` and `resume` calls retain their usual meanings.

The core philosophy here is that these functions that are common to all kinds of devices. It is the job of every device driver to provide implementations for these functions. Creating such a structure with function pointers is a standard design technique – it is similar to virtual functions in C++ and abstract functions in Java.

A Generic Device

Listing 7.3: `struct device`
source : include/linux/device.h

```
struct device {
    /* generic information */
    dev_t devt;
    u32 id;

    /* parent device, bus and device driver */
    struct device *parent;
    struct bus_type *bus;
    struct device_driver *driver;

    /* Physical location of the device */
    struct device_physical_location *physical_location;

    /* DMA-related fields */
    struct bus_dma_region *dma_range_map;
    struct list_head dma_pools;
}
```

The code of `struct device` is shown in Listing 7.3. Every device contains a $\langle major, minor \rangle$ number pair (`devt`) and an unsigned 32-bit id.

Devices are arranged as a tree. Every device thus has a parent. It additionally has a pointer to the `bus` and its associated device driver. Note that a device driver does not point to a device because it can be associated with many devices. Hence, the device is given as an argument to the functions defined in the device driver. However, every device needs to maintain a pointer to its associated device driver because it is associated with only a single one.

Every block device has a physical location. There is a generic way of describing a physical location at which the block device is connected. It is specified using `struct device_physical_location`. Note the challenges in designing such a data structure. Linux is designed for all kinds of devices: wearables, mobile phones, laptops, desktops and large servers. There needs to be a device-independent way for specifying where a device is connected. The kernel defines a location panel (id of the surface on the housing), which can take generic values such as top, left, bottom, etc. A panel represents a generic region of a device. On each panel, the horizontal and vertical positions are specified. These are coarse-grained positions: (top, center, bottom) and (left, center, right). We additionally store two bits. One bit indicates whether the device is connected to a docking station and the second bit indicates whether the device is located on the lid of the laptop.

Block devices often read and write large blocks of data in one go. Port-mapped I/O and memory-mapped I/O often turn out to be quite slow and unwieldy in such cases. DMA-based I/O is much faster in this case. Hence, every block I/O device is associated with a DMA region. Further, it points to a linked list of DMA pools. Each DMA pool points to a set of buffers that can be used for DMA transfers. These are buffers in kernel memory and managed by a slab cache (refer to Section 6.4.2).

A Block Device and a Generic Disk

Listing 7.4: `struct block_device`
source : include/linux/blk_types.h

```
struct block_device {
    /* pointer to the encompassing device data structure */
    dev_t bd_dev;
    struct device bd_device;

    /* starting sector and number of sectors */
    sector_t bd_start_sect;
    sector_t bd_nr_sectors;

    /* generic disk and request queue */
    struct gendisk *bd_disk;
    struct request_queue* bd_queue;

    /* file system related fields */
    struct super_block *bd_super;
    struct inode *bd_inode;
}
```

The code of a block device is shown in Listing 7.4. It is like a derived class where the base class is a `device`. Given that C does not allow inheritance, the next best option is to add a pointer to the base class (`device` in this case) in the definition of `struct block_device`. Along with a pointer, we add the version numbers as well in the device type (`devt`) field.

Every block device is divided into a set of sectors. However, it can be divided into several smaller devices that are *virtual*. Consider a hard disk, which is a block device. It can be divided into multiple partitions. For example, in Windows they can be `C:`, `D:`, `E:`, etc. In Linux, popular partitions are `/swap`, `/boot` and the base directory '`/`'. Each such partition is a virtual disk. It represents a contiguous range of sectors. Hence, we store the starting sector number and the number of sectors.

For historical reasons, a block device is always associated with a generic disk. This is because the most popular block devices in the early days were hard disks. This decision has persisted even though there are many more types of block devices these days such as SSD drives, NVM memories, USB storage devices, SD cards and optical drives. Nevertheless, a block device structure has a pointer to a `struct gendisk`.

Listing 7.5: `struct gendisk`
source : include/linux/blkdev.h

```
struct gendisk {
    /* major device number and name of the disk */
    int major;
    char disk_name[];

    /* table of partitions */
    struct block_device *part0;
    struct xarray part_tbl;     /* partition table */
```

```
    /* table of function pointers */
    struct block_device_operations *fops;

    /* pointer to a request queue */
    struct request_queue *queue;
}
```

The definition of `struct gendisk` is shown in Listing 7.5. Along with a major number and name, the key data structures are a pointer to the associated block device and a table of partitions. The notion of partitions is an integral part of the definition of a generic disk. These represent storage devices, which often have partitions regardless of the type of the device. The raw block device can be thought of as a wrapper of the default partition (partition 0).

The partition table `part_tbl` manages partitions *dynamically*. Each entry contains the details of each partition: starting sector, size and metadata. Each entry in the partition table is a pointer to block device structure (`struct block_device`). Recall that we had associated a block device structure with each partition.

The next important data structure is a pointer to a structure called `block_device_operations`. It contains a set of function pointers that are associated with different functions that implement specific functionalities. There are standard functions to open a device, release it (close it), submit an I/O request, check its status, check pending events, set the disk as read-only and freeing the memory associated with the disk.

Let us now discuss the request queue that is a part of the `gendisk` structure. It contains all the requests that need to be serviced.

The Request Queue

Listing 7.6: `struct request_queue`
source : include/linux/blkdev.h

```
struct request_queue {
    /* pointer to the last request */
    struct request *last_merge;

    /* I/O request queue that interfaces with the I/O
       scheduler */
    struct elevator_queue *elevator;

    /* per-CPU software request queue */
    ...

    /* per-device request queues */
    ...
}
```

The code of `struct request_queue` is shown in Listing 7.6. It stores a small amount of current state information – the last request that has been serviced (`last_merge`).

The key structure is a queue of requests – an elevator queue. Let us explain the significance of the *elevator* here. We need to understand how I/O requests

are scheduled in the context of storage devices, notably hard disks. We wish to minimize the seek time (refer to Section 7.2.1). The model is that at any point of time, a storage device will have multiple pending requests. They need to scheduled in such a way that per-request the disk head moves the least. One efficient algorithm is to schedule I/O requests the same way an elevator schedules its stops. We will discuss more about this algorithm in the section on I/O scheduling algorithms.

The two other important data structures that we need to store are I/O request queues. The first class of queues are per-CPU software request queues. They store pending requests for the I/O device. It is important to note that these are waiting requests that have not been scheduled to execute on the storage device yet. Once they are scheduled, they are sent to a per-device request queue that sends requests directly to the underlying hardware. These per-CPU queues are lockless queues, which are optimized for speed and efficiency. Given that multiple CPUs are not accessing them at the same time, there is no need for locks and other concurrency control mechanisms. At this point, it is possible to merge and reorder requests. This will make I/O processing more efficient. For example, if there are multiple writes to the same disk block, then the write requests can be merged. A later read can be reordered to appear before an earlier write (to a different address). This is because read requests are often on the critical path. This queue structure is `struct blk_mq_ctx` in the current version of the kernel.

Note that most storage devices have internal request queues implemented at the hardware level. They store pending I/O requests. The per-device request queue (on the other hand) maintains requests that need to be sent to the device, which basically means that a request leaves the queue and enters the hardware-level request queue. This request is subsequently serviced by the device. Such a process of sending requests from the per-device software queue to the hardware's queue is known as *syncing* (short form for synchronizing). This needs to be done regularly and periodically. The frequency depends upon the overheads, the request rate and the speed of the device. If requests are sent too eagerly, then a lot of CPU time will be lost in this process. If they are sent less aggressively, then the I/O response latency will be low. This queue is implemented by `struct blk_mq_hw_ctx`.

Let us end this discussion with a short discussion on operational aspects. The per-device queue (also known as the dispatch queue) has requests from all CPUs. It is important to identify every request and its corresponding response with a tag. This is needed to match requests and their respective responses.

The request queues are shown in Figure 7.22. We can see two types of queues: HW queues (queues that are periodically synced with the device request queues) and pure software queues that store requests that are yet to be scheduled. Let us now look at each entry of these queues. Each such entry needs to store an I/O request (struct request). Such an I/O request is represented by a `struct request` (shown in Listing 7.7).

It stores a few pointers to associated data structures such as the request queue, software and hardware queues and the block device. The next few fields store the details of the I/O request. Some such fields of our interest are the total length of the data (data length), starting sector number, the deadline and a timeout value (if any). The fact that block I/O requests can have a deadline associated with them is important. This means that they can be treated as soft
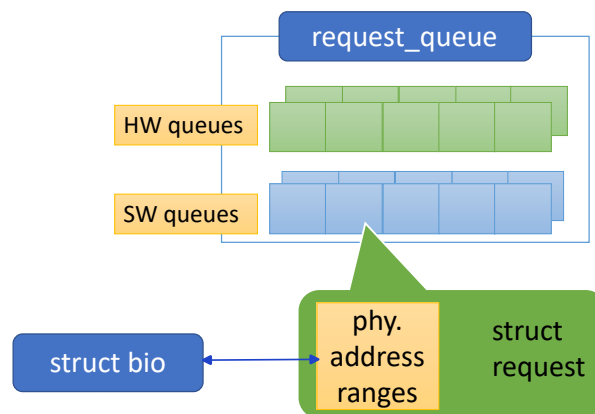
Figure 7.22: Request queues in the device driver subsystem

real time tasks.

Listing 7.7: `struct request`
source : include/linux/blk − mq.h

```c
struct request {
    /* Back pointers */
    struct request_queue *q;
    struct blk_mq_ctx *mq_ctx;
    struct blk_mq_hw_ctx *mq_hctx;
    struct block_device *part;

    /* Parameters */
    unsigned int __data_len;
    sector_t sector;
    unsigned int deadline, timeout;

    struct bio *bio;
    rq_end_io_fn *end_io;
}
```

There are two more fields of interest. The first is a function pointer (`end_io`) that is invoked to complete the request. This is device-specific and is implemented by its driver code. The other is a generic data structure that has more details about the I/O request (`struct bio`).

Listing 7.8: `struct bio`
source : include/linux/blk_types.h

```c
struct bio {
    struct block_device    *bi_bdev;
    struct bio_vec         *bi_io_vec;
}
```

Its structure is shown in Listing 7.8. It has a pointer to the block device and an array of memory address ranges (`struct bio_vec`). Each entry is a 3-tuple: physical page number, length of the data and starting offset. It points to a

memory region that either needs to be read or written to. A `bio_vec` structure is a list of many such entries. We can think of it as a sequence of memory regions, where each single chunk is contiguous. The entire region represented by `bio_vec` however may not be contiguous. Moreover, it is possible to merge multiple `bio` structs or `bio_vec` vectors to create a larger I/O request. This is often required because many storage devices such as SSDs, disks and NVMs prefer long sequential accesses.

7.4.4 I/O Scheduling

Let us now discuss I/O scheduling algorithms. We have already looked at the classical elevator scheduling algorithm. In this case, the disk head moves from the innermost track to the outermost track servicing requests on the way. It is possible to optimize this process by starting from the request that is closest to the center (innermost) and stop at the request that is the farthest from the center (outermost). This minimizes back and forth movement of the disk head, and also ensures fairness. After reaching the outermost request, the disk head then moves towards the innermost track servicing requests on they way. An elevator processes requests in the same manner.
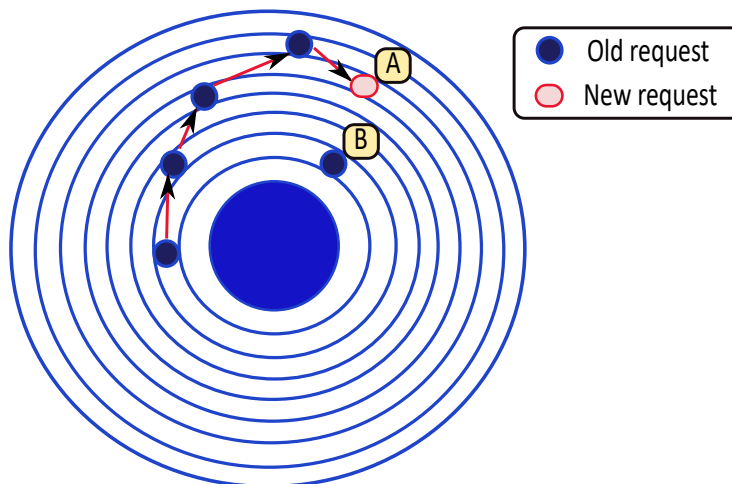
Figure 7.23: Example of the elevator algorithm, where fairness is being compromised. Fairness would require Request B to be scheduled before Request A because it arrived earlier. If we start servicing requests on the reverse path (outer to inner) then all the requests in the vicinity (nearby tracks) of A will get serviced first. Note that requests in the vicinity of A got two back-to-back chances: one when the head was moving towards the outermost track and one when it reversed its direction.

There are many variants of this basic algorithm. We can quickly observe that fairness is slightly being compromised here. Assume the disk head is on the track corresponding to the outermost request. At that point of time, a new request arrives. It is possible for the disk head to immediately reverse its direction and process the new request that has just arrived. This will happen

if it is deemed to be in the outermost track (after the earlier request has been processed). This situation is shown in Figure 7.23.

It is possible to make the algorithm fairer by directly moving to the innermost request after servicing the outermost direction. In the reverse direction (outer to inner), no requests are serviced. It is a direct movement of the head, which is a relatively fast operation. These classes of algorithms are very simple and not used in modern operating systems.

Linux uses three I/O scheduling algorithms: Deadline, BFQ and Kyber.

Deadline Scheduler

The Deadline scheduler stores requests in two queues. The first queue (sorted queue) stores requests in the order of their block address, which is roughly the same as the order of sectors. The reason for such a storage structure is to maximize contiguous accesses without incurring overheads associated with the seek time and rotational delay. This is the preferred method. Additionally, this scheduler prefers reads over writes because they are typically on the critical path. The scheduler maintains one more queue (deadline queue), where requests are sorted by their deadline. If a request gets delayed significantly and its deadline has expired, it is scheduled immediately. In this case, entries in the deadline queue take precedence over entries in the sorted queue. This design ensures high-performance I/O and at the same time guarantees a bound on the latency.

BFQ Scheduler

The BFQ (Budget Fair Queuing) scheduler is similar to the CFS scheduler for processes (see Section 5.4.6). The same way that CFS apportions the processing time between jobs, BFQ creates sector slices and gives every process the freedom to access a certain number of sectors in the sector slice. The main focus here is fairness across processes. Latency and throughout are secondary considerations.

Kyber Scheduler

This scheduler was introduced by Facebook (Meta). It is a simple scheduler that creates two buckets: high-priority reads and low-priority writes. Each type of request has a target latency. Kyber dynamically adjusts the number of allowed in-flight requests such that all operations complete within their latency thresholds.

General Principles

In general, I/O schedulers and libraries perform a combination of three operations: delay, merge and reorder. Sometimes it is desirable to delay requests a bit such that a set of sufficient size can be created. It is easy to apply optimizations on such a set with a sizable number of requests. One of the common optimizations is to merge requests. For example, reads and writes to the same block can be easily merged, and redundant requests can be eliminated. Accesses to adjacent and contiguous memory regions can be combined. This will minimize the seek time and rotational delay.

Furthermore, requests can be reordered. We have already seen examples of reordering reads and writes. This is done to service reads quickly because they

are often on the critical path. We can also distinguish between synchronous writes (we wait for it to complete) and asynchronous writes. The former should have a higher priority because there is a process that is waiting for it to complete. Other reasons for reordering could factor in the current position of the disk head and deadlines.

7.4.5 A Simple Block Device Driver

Let us look at a simple device driver. Sony's memory stick driver is a quintessential example (/drivers/memstick/core/mspro_block.c).

The module starts by calling the function `mspro_block_init`. It registers the driver's name and the block device "mspro_block". As we have seen earlier, every driver is a structure that contains a set of function pointers.

In this case, the structure that represents the driver is `mspro_block_driver`. It contains pointers to the `probe`, `initialize`, `remove`, `suspend` and `resume` functions.

The initialization function `mspro_block_probe` initializes the memory card. It sends it instructions to initialize its state and prepare itself for subsequent read/write operations. Next, it creates an entry in the `sysfs` file system, which is a special file system that exposes attributes of kernel objects such as devices to users. Files in the `sysfs` file system can be accessed by user-level applications to find the status of devices. In some cases, superusers can also write to these files, which allows them to control the behavior of the corresponding devices. Subsequently, the initialization function initializes the various block device-related structures: `gendisk`, `block_device`, `request_queue`, etc.

Typically, a device driver is written for a family of devices. For a specific device in the family, either a generic (core) function can be used or a specific function can be implemented. Let us consider one such specific function for the Realtek USB memory card. It uses the basic code in the `memstick` directory but defines its own function for reading/writing data. Let us explain the operation of the function `rtsx_usb_ms_handle_req`.

It maintains a queue of outstanding requests. It uses the basic `memstick` code to fetch the next request. There are three types of requests: read, write and bulk transfer. For reading and writing, the code creates generic USB commands and passes them on to a low-level USB driver. Its job is to send the raw commands to the device. For a bulk transfer, the driver sets up a one-way pipe with the low-level driver, which ensures that the data is directly transferred to a memory buffer, which the USB device can access. The low-level commands can be written to the USB command registers by the USB driver.

Point 7.4.1

1. Most of the work involves framing appropriate requests and optimizing traffic to the device. This involves various operations that rely on smart scheduling, delaying, merging and reordering requests.

2. The code for communicating with the device is a small part of the overall driver code.

> 3. Several drivers often cooperate to perform a task. The core driver provides generic functionalities for a family of devices. Device-specific drivers may override some of that functionality. Drivers often take the help of other low-level drivers to communicate with devices, especially when multiple protocols are involved. For example, in this case, the memory stick driver needed the help of USB drivers to communicate with the memory stick over USB.

7.5 Character Devices

Character device drivers are comparatively much simpler. Their latency and throughput constraints are more relaxed. Hence, they can be implemented as modules, and their implementations can be less complicated.

Let us consider the USB keyboard driver (/drivers/hid/usbhid/usbkbd.c). The entry point of the module is the function `usb_kbd_probe`. Its job is to allocate a generic input device and associate the keyboard device with it. Akin to block devices, there are two functions to register and deregister character devices (resp.). There are three function pointers that need to be initialized. ① A function that is called when the module is loaded. ② A function that is called when the module is unloaded or the device is ejected. ③ A function that is invoked when there is a "key press" event.

A character device driver does not have to do the complicated request processing that block device drivers need to do. In their case, they just need to establish a connection between the interrupt and the function implemented by the device driver. Whenever a key is pressed, an interrupt is raised. The default interrupt handling system can find the device that has raised the interrupt and call its IRQ handler function. This is quite an involved process as we have seen in Chapter 4. It is possible that multiple devices are sharing the same IRQ line. This makes it necessary to query each device and find if it is the one that had raised the interrupt. The mechanisms discussed in Chapter 4 ensure that all of this is seamlessly handled. We finally end up calling the function associated with the device.

This device could be a composite device such as a USB controller, which is connected to multiple ports. Each port may be connected to a hub that can be attached to many more USB devices. Only one IRQ number is associated with the USB controller. Its job is to identify each connected USB device and assign it a unique USB address, which is internal to the USB subsystem. Hence, the situation can be summarized as follows. The generic code of the kernel identifies the USB controller as the interrupting device when we press a key on a USB keyboard. The controller's interrupt handler further queries the controller and finds the address of the specific device. In this case, it is the address of the USB keyboard. The device driver starts directly communicating with the keyboard via the controller. It reads the data at this point, which comprises all the information regarding the keys that were pressed.

Let us delve into the specifics. The USB keyboard registers a function called `usb_kbd_irq` with the kernel (refer to Listing 7.9). The generic USB device driver invokes it, when it gets information from the USB keyboard regarding

the interrupt.

<div align="center">

Listing 7.9: <code>usb_kbd_irq</code> function

source : drivers/hid/usbhid/usbkbd.c

</div>

```
void usb_kbd_irq(struct urb *urb);
```

The argument is a `struct urb`, which is a generic data structure that holds information pertaining to USB requests and responses. It holds details about the USB endpoint (device address), status of the transfer, pointer to a memory buffer that holds the data and the type of the transfer. The details of the keys that were pressed are present in this memory buffer. Specifically, the following pieces of information are present for keyboards.

1. Status of special keys: Ctrl, Alt, Scroll Lock

2. Check if the same key has been continuously pressed.

3. The character corresponding to the key that was pressed (internal code or ASCII code).

4. Report whether Num Lock or Caps Lock have been processed.

The final output is either an ASCII character or some interrupt.

Point 7.5.1

Unlike block devices that have direct connections to the motherboard's buses, character devices are connected to the motherboard via a network of ports and controllers. The latency and throughput constraints are more relaxed for such devices. The main challenge in managing such devices is to effectively ferry the data across various buses, controllers, ports, protocols and device drivers. A long chain of callback functions needs to be created and data needs to be efficiently passed between the device and its driver. Given that such devices are often hot-pluggable, the driver and OS utilities need to be quite responsive.

7.6 File Systems

Let us now look at the design of a regular file system. We shall delve into the details of the everything-is-a-file assumption, and instead focus on conventional files and directories. Insofar, as the OS is concerned, a file is an array of bytes, or equivalently an array of logical blocks. These blocks are stored on a storage device.

7.6.1 Tree-Structured Layout of a File System

A file system organizes the files and directories in a classical tree-structured layout (see Figure 7.24). The directories (folders) are the internal nodes, and the leaves are regular data files. In reality, a directory is a special kind of data file; its contents represent the contents of the corresponding folder. It stores a table, where each row represents the name of a file or subdirectory

in the folder. Specifically, the following fields are stored: name, permissions, ownership information and a pointer to its metadata. The summary of this discussion is that we treat a directory as a data file that stores a simple table. Henceforth, we shall use the term "file" to refer to a generic file and the term "regular file" to refer to a file that just stores data.
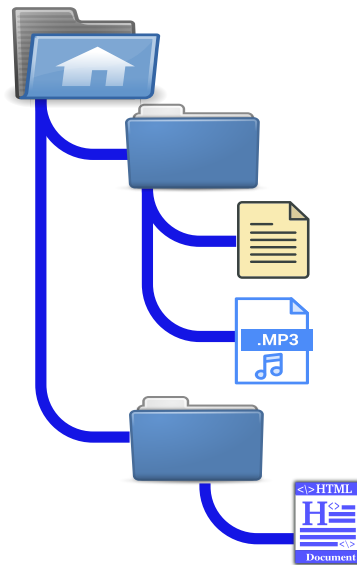
Figure 7.24: Tree-structured organization of a file system.

The role of the metadata associated with each file (regular or directory) is primarily to map logical addresses to physical addresses in the storage device. Every file is assumed to start at logical address zero. If its size is n bytes, then the last logical address is $n - 1$. However, these bytes can seldom be stored contiguously on disk because of internal and external fragmentation. Recall that we had seen similar problems in the case of physical memory as well (see Section 6.1.1). This has necessitated virtual memory. Therefore, akin to virtual memory, there is a need to create a translation layer here as well. It needs to map logical addresses (within files) to physical addresses (within a storage device). There is thus a need to create a structure similar to a page table. Note that this is at the level of each and every file, it is not a centralized structure. Most Linux-like operating systems define the concept of an *inode* (see Figure 7.25). It stores the metadata associated with a file like its name, ownership information, size, permissions, etc., and also has a pointer to the *block mapping table*. If the user wishes to read the 1046th byte of a file, all that she needs to do is compute the block number and pass the file's inode to a generic function. The outcome is the address on the storage device.

Now a directory is also in a certain sense a regular file that stores data. It is thus also represented by an inode. Since an inode acts like a metadata storage unit in the world of files, it does not care what it is actually representing. It can represent either regular files or directories or even devices. While representing data (files and directories), it simply stores pointers to all the constituent blocks without caring about their semantics. A block is treated as just a collection of
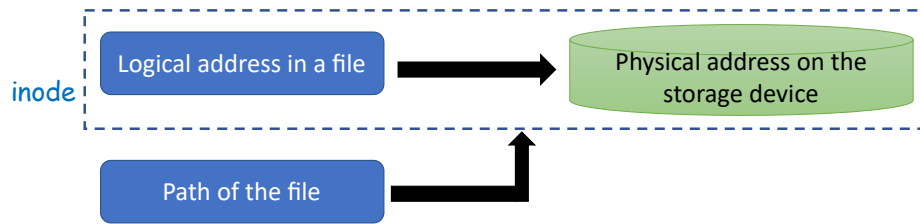
Figure 7.25: Notion of a mapping table (either pointed to by an inode or stored within it).

bytes. A directory's structure is also simple. It stores a table that is indexed by the name of the file/directory. The columns store the metadata information. One of the most important fields is a pointer to the inode of the entry. This is the elegance of the design. The inode is a generic structure that can point to any type of file including network sockets, devices and inter-process pipes – it does not matter.

Let us explain with an example. In Linux, the default file system's base directory is `/`. Every file has a path. Consider the path `/home/srsarangi/ab.txt`. Assume that an editor wants to open the file. It needs to access its data blocks and thus needs a pointer to its inode. The `open` system call locates the inode and provides a handle to it that can be used by user programs. Assume that the location of the inode of the `/` (or root) directory is known. It is inode #2 in the ext4 file system. The kernel code reads the contents of the `/` directory and locates the inode of the `home` subdirectory in the table of file names. This process continues recursively until the inode of `ab.txt` is located. Once it is identified, there is a need to remember this information. The inode is wrapped in a file handle, which is returned to the process. For subsequent accesses such as reading and writing to the file, all that the kernel needs is the file handle. It can easily extract the inode and process the request. There is a no need to recursively traverse the tree of directories.

Let us now look at the file system in its entirety. It clearly needs to store all the constituent inodes. Let us look at the rest of its components.

Recall that in hard disks and similar block devices, a single physical device can be partitioned into multiple logical devices or logical disks. This is done for effective management of the storage space, and also for security purposes – we may want to keep all the operating system related files in one partition and store all the user data in another partition. Some of these partitions may be bootable. Bootable partitions typically store information related to booting the kernel in the first sector (Sector 0), which is known as the *boot block*. The BIOS can then load the kernel.

Most partitions just store a regular file system and are not bootable. For example, D: and E: are partitions on Windows systems (refer to Figure 7.24). On Linux, `/usr` and `/home` may be mounted on different partitions. In general, a partition has only one file system. However, there are exceptions. For example, `swap` on Linux (swap space) does not have a file system mounted on it. There are file systems that span multiple partitions, and there are systems where multiple file systems are mounted on the same partition. However, these are

very specialized systems. The metadata of most file systems is stored in Block 1, regardless of whether they are bootable or not. This block is known as the *superblock*. It contains the following pieces of information: file system type and size, attributes such as the block size or maximum file length, number of inodes and blocks, timestamps and additional data. Some other important data structures include inode tables, and a bitmap of free inodes and disk blocks. For implementing such data structures, we can use bitmaps that can be accelerated with augmented trees.

7.6.2 Mounting a File System

Let us now explain what it means to *mount a file system*. A file system is a tree-structured data structure. It is shown as a triangle in Figure 7.26. Most operating systems have a default file system, which stores the system-wide root directory. In Windows, it is "My Computer", and in Linux it is `/`. Assume a Linux system, where we insert a new storage device. A file system needs to be mounted on it. It will have its dedicated superblock and inodes, as well as a root directory. The key question is how do we access the files stored in the new file system? Any file or directory has a path that is of the form `/dir1/dir2/.../filename` in Linux. In Windows `/` is replaced with `\`. The baseline is that all files need to be accessible via a string of this form, which is known as the *path of the file*. It is an absolute path because it starts from the root directory. A path can also be <u>relative</u>, where the location is specified with respect to the current directory. Here the parent directory is specified with the special symbol "..". The first thing that the library functions do is convert all relative paths to absolute paths. Hence, the key question still remains. How do we specify paths across file systems?
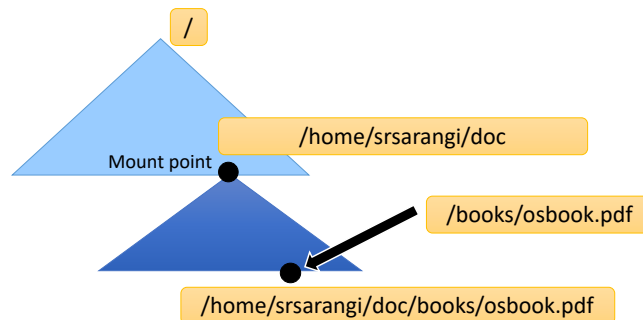
Figure 7.26: Mounting a file system

Consider the example in Figure 7.26 again. We need to attach a new file system to the root file system (one that contains the `/` directory). The way to do this is to create a dummy directory in the root file system and make that the root directory of the new file system. Let the dummy directory be `/home/srsarangi/doc`. Assume that your author's documents directory is mounted on the new storage device. The mounting process makes the dummy directory the root directory of the mounted file system. This is the <u>mount point</u>. In the file system on the storage device, consider a file at the

location `/books/osbook.pdf`. Once this file system is mounted, then in the unified file system, its new location is `/home/srsarangi/doc/books/osbook.pdf` (refer to Figure 7.26).

The `mount` command in Linux is used to mount a file system. Whenever we insert a pen drive, the system automatically mounts its root file system. The root directory of the USB file system gets mapped to a directory on the host machine's file system. For example, it can become a directory such as `/mnt/usb`. The file `/videos/foo.mpg` in the USB's file system becomes `/mnt/usb/videos/foo.mpg`. The advantage of mounting a file system in this manner is that all the files in the system have a common naming and addressing system across file systems. All absolute paths have the same basic format.

The next logical question is how does the kernel figure out how to process files, especially when they are in different file systems. Let us consider the path `/mnt/usb/foo.mpg` again. `/mnt` is a part of the root file system. The default file system driver can traverse the file system and reach the mount point /mnt/usb. It will then realize that it is a mount point, and the mounted file system is different. From this point onwards, it will invoke the driver of the mounted file system. It will be tasked with retrieving the file `/videos/foo.mpg` relative to its root. Things can get more interesting. A mounted file system can mount another file system, so on and so forth. The algorithm for traversing the file system remains the same. Recursive traversal involves first identifying the file system from the file path, and then invoking its functions to locate the appropriate inode.

Finally, the `unmount` command can be used to unmount a file system. Its files will not be accessible anymore.

7.6.3 Soft Links and Hard Links
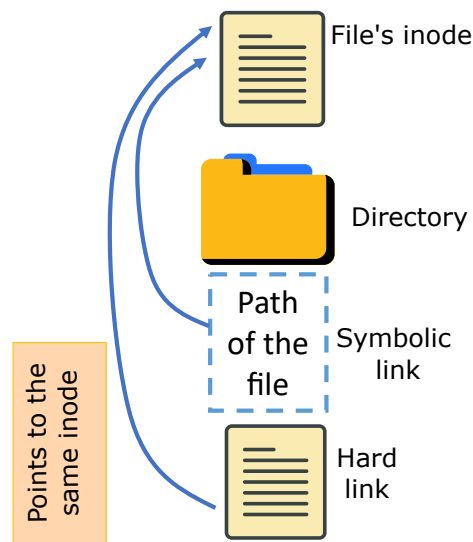
Figure 7.27: Hard and soft links

Soft Links or Symbolic Links

Let us now slightly tinker with the tree structure and create a DAG. This is sometimes useful, especially when we want to create shortcuts. Consider a path that is very long. The user may not always want to type such a long pathname. It is a much better idea to create a shortcut. For example, `/home/srsarangi/docs` can be made to point to
`/home/srsarangi/Documents/september/monday/noon/alldocs/pdfs`. Such shortcuts are very convenient. They are known as symbolic links or soft links in Linux. It is very easy to create it (refer to Listing 7.10).

Listing 7.10: Creating a soft/symbolic link

```
ln -s path_to_target path_to_link
```

A separate file is created with its own inode (file type 'l'). Its contents contain the path of the target. Hence, resolving such a link is straightforward. The kernel reads the path contained in the symbolic link file, and then uses the conventional algorithm to identify its inode. A symbolic link makes working with the file system easier. However, it does not introduce additional efficiency. If the link is deleted, the target is unaffected. If the target is deleted, then the link becomes useless.

This mechanism sadly does not solve all our problems. Assume a scenario where we would like to take backups. The aim is to create a new directory that contains links to only those files that have been recently updated. Assume we have some algorithm to identify these files. The aim is to minimize storage space overheads. We thus cannot copy the files. Even symbolic links waste space in terms of storing the full paths. Furthermore, a symbolic link is very strongly tied to the path of the original file. The latter cannot be moved. It is possible that the original target file is deleted, and another file with the same name is created. The link will still work. This is quite problematic.

Hard Links

Hence, hard links were introduced. The same `ln` command can be used to create hard links as follows (refer to Figure 7.27).

Listing 7.11: Creating a hard link

```
ln  path_to_target path_to_link
```

A hard link is a directory entry that points to the same inode as the target file. In this case, both the hard link and the directory file point to the same inode. If one is modified, then the changes are reflected in the other. However, deleting the target file does not lead to the deletion of the hard link. Hence, the hard link still remains valid. We pretty much maintain a reference count with each inode. The inode is deleted when all the files and hard links that point to it are deleted. Another interesting property is that if the target file is moved (within the same file system) or renamed, the hard link still remains valid. However, if the target file is deleted and a new file with the same name is created in the same directory, the inode changes and the hard link does not remain valid.

There are nonetheless some limitations with hard links. They cannot be used across file systems, and normally cannot link directories. The latter will create infinite loops because a child can now link to an ancestor directory.

7.6.4 Virtual File System
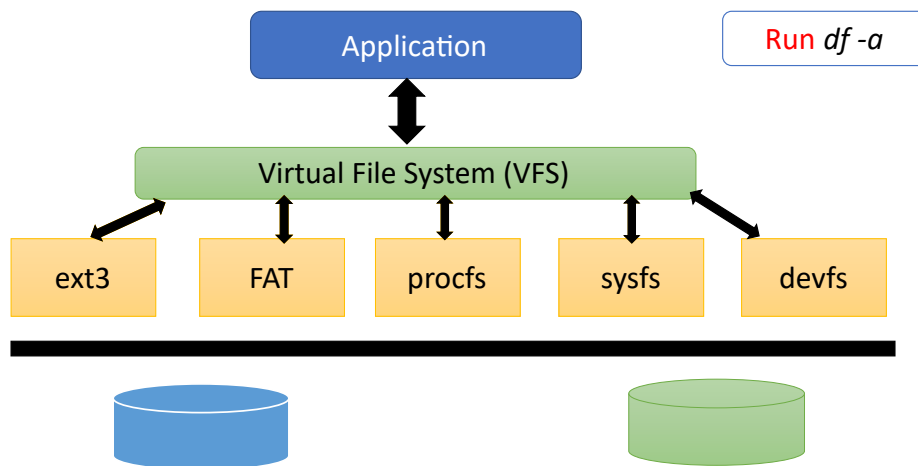
Figure 7.28: File systems supported by the Linux virtual file system (VFS).

Let us now tackle a fundamental issue that we have conveniently ignored up till now. We have assumed a unified file system that can tie together many file systems, and create a single tree-structured namespace for all the files and directories in the system. Needless to say, this is the very beneficial for users who are using the unified file system. They need not bother about mount points and the details of the underlying file systems. Insofar as they are concerned, the entire system has a single "virtual file system" (VFS). Any file in this virtual file system has a path that is in the standard format (uses '/' as the delimiter), and can be processed using regular system calls such as `open`, `close`, `read`, `write`, etc.

A virtual file system (VFS) is like virtual memory, which conveniently abstracts out the details of the underlying technology and hardware. Figure 7.28 shows a conceptual view of the virtual file system where a single file system unifies many different types of file systems. We wish to use a single interface to access and work with all the files in the VFS regardless of how they are stored or which underlying file system they belong to. Finally, given Linux's historical ties to Unix, we would like the VFS's interface to be similar to that of the classical Unix file system (UFS). Given our observations, let us list down the requirements of a virtual file system in Point 7.6.1.

> **Point 7.6.1**
>
> 1. The virtual file system (VFS) should be similar in terms of its interface to the classical Unix file system (UFS). This includes the functions used to work with files, file metadata and data structures

(inodes, directory entries and superblocks).

2. Regardless of the number and type of mounted file systems, we wish to have a single tree-structured file system tree (without considering links). The internal nodes are the directories and the leaves are the files.

3. The root directory is '/'. Hard links and symbolic links are supported.

4. File systems can be mounted at any point in the directory tree.

5. The everything-is-a-file assumption is followed. This means that regular files, directories, devices, sockets, pipes, etc., are represented as VFS files. They can be accessed and processed using standard UFS file system functions.

Virtualization of File Systems

Figure 7.28 shows a few file systems whose details are conveniently virtualized by VFS. The ext3 and ext4 file systems are the default file systems on most Linux systems as of today. The FAT (File Allocation Table) file system is used in USB drives and in embedded devices. `procfs` is a special file system that shows information about running processes, status of memory modules, CPU usage and kernel data structures. It is a mechanism for the kernel to expose information to user processes. For example, the file `/proc/meminfo` shows the details of memory usage and `/proc/cpuinfo` stores all CPU-related information.

The `sysfs` file system is mounted at `/proc/sysinfo`. It stores information about devices, drivers, buses, file systems and kernel data structures. Each directory in `/sys` corresponds to a kernel object – a software representation of devices, drivers and important kernel subsystems. On similar lines, the `devfs` file system is mounted on `/dev`. This directory stores multiple files (one file per device). Some modern device managers such as `udev` dynamically create and destroy device files as and when they are plugged and unplugged.

The greatness of VFS is that it virtualizes all these file systems and creates one common standard interface. The `df` command can be used to find the file systems mounted on a standard Linux system.

Creation of Dummy inodes and Directory Entries

Now, let us consider file systems such as `devfs` or NFS (network file system) that do not use inodes. Users or even other parts of the kernel should obviously be unaware of the semantics of underlying file systems. Hence, they are all virtualized by VFS, which exposes standard structures such as inodes, directory entries, superblocks, file objects and page caches to other kernel subsystems. Even if the file system does not support inodes or superblocks, other kernel subsystems need to see a view of it that exposes these data structures.

This is easily achieved by the virtualization layer. If the underlying file system provides these structures, then all that it needs to do is appropriately wrap them and expose them to the rest of the kernel. However, if they are not

provided, then there is a need to create pseudostructures. A "pseudostructure" is for example a dummy inode. It is created by VFS when a file is accessed. VFS checks whether the underlying file system supports inodes or not. If it does not, then it creates a dummy or pseudoinode structure and caches it. It returns a pointer to this inode to any kernel function that wishes to work with the file. Such a kernel function is blissfully unaware of the fact that it is actually dealing with a psuedoinode. It invokes all the functions that it would invoke on regular inodes. It is the job of VFS to appropriately translate these calls and forward them to the driver of the underlying file system. The return value is also suitably modified. VFS thus acts as an intermediary between the real file system and kernel subsystems. It can make all file systems *appear* to be a classical Unix file system. This makes interfacing with the kernel significantly easier.

To do this, it is necessary to maintain a collection of pseudostructures including pseudoinodes, pseudodirectory entries, pseudosuperblocks and so on. Many such structures can be created when a file or directory entry is accessed for the first time. Some more such as the pseudosuperblock need to be created when the file system is mounted. Then they can be added to a software cache. Any subsequent access will find the pseudostructure in the cache.

7.6.5   Structure of an inode

Listing 7.12 shows the key fields of `struct inode`. Every inode contains a pointer to a superblock. As we have discussed earlier, the superblock maintains important information about the entire file system. Every inode has a unique inode number. This is sometimes used to identify it.

The page cache is an important data structure. Its role is to cache pages that are stored on a storage device such as a hard disk. Most of the I/O accesses to the storage device are actually served by the in-memory page cache. This makes I/O operations significantly faster. Of course, the price that is paid is consistency. If there is a power failure, then a lot of data that is not written to permanent storage ends up getting lost. This is nevertheless an acceptable risk given the huge performance improvement. It is possible to manually synchronize the contents with the underlying storage device using the `sync` function call. There are several options for synchronizing files: periodically synchronize them, synchronize when a file is closed or synchronize when the file system is unmounted. Based on the configuration user applications can be written such that there is no violation in correctness (keeping the synchronization strategy in mind). Every inode points to the corresponding page cache (field: `i_data`). Along with a pointer to the page cache, an inode additionally stores a pointer to the device that hosts the file system (`i_rdev`).

<div align="center">

Listing 7.12: `struct inode`

source : include/linux/fs.h

</div>

```
struct inode {
    /* Pointer to the superblock */
    struct super_block    *i_sb;

    unsigned long          i_ino;  /* unique inode number */
    struct address_space  *i_data; /* ptr to page cache */
```

```
    dev_t              i_rdev;            /* ptr to device */

    /* size of the file */
    blkcnt_t           i_blocks;
    loff_t             i_size;

    /* ownership and permission information */
    umode_t            i_mode;
    kuid_t             i_uid;
    kgid_t             i_gid;

    /* point to a set of functions */
    const struct inode_operations   *i_op;

    /* pointer to a file system or device */
    void               *i_private;
}
```

Next, we store the size of the file in terms of the number of blocks (`i_blocks`) and the exact size of the file in bytes (`i_size`).

We shall study in Chapter 8 that permissions are very important in Linux from a security perspective. Hence, it is important to store ownership and permission information. The field `i_mode` stores the type of the file. Linux supports several file types namely a regular file, directory, character device, block device, FIFO pipe, symbolic link and socket. Recall that everything-is-a-file assumption. The file system treats all such diverse entities as files. Hence, it becomes necessary to store their type as well. The field `i_uid` shows the id of the user who is the owner of the file. In Linux, every user belongs to one or more groups. Furthermore, resources such as files are associated with a group. This is indicated by the field `i_gid` (group id). Group members get some additional access rights as compared to users who are not a part of the group. Some additional files include access times, modification times and file locking-related state.

The next field `i_op` is crucial to implementing VFS. It is a pointer to an inode operations structure that contains a list of function pointers. These function pointers point to generic file operations such as open, close, read, write, flush (move to kernel buffers), sync (move to disk), seek and mmap (memory map). Note that each file system has its own custom implementations of such functions. The function pointers point to the relevant function (defined in the codebase of the underlying file system).

Given that the inode in VFS is meant to be a generic structure, we cannot store more fields. Many of them may not be relevant to all file systems. For example, we cannot store a mapping table because inodes may correspond to devices or sockets that do not store blocks on storage devices. Hence, it is a good idea to have a pointer to data that is used by the underlying file system. The pointer `i_private` is useful for this purpose. It is of type `void *`, which means that it can point to any kind of data structure. Often file systems set it to custom data structures. Many times they define other kinds of encapsulating data structures that have a pointer to the VFS inode and file system-specific custom data structures. `i_private` can also point to a device that corresponds to the file. It is truly generic in character.

> **Point 7.6.2**
>
> An inode is <u>conceptually</u> a two-part structure. The first part is a VFS inode (shown in Listing 7.12), which stores generic information about a file. The second part is a file system-specific inode that may store a mapping structure, especially in the case of regular files and directories.

Directory Entry

Listing 7.13: `struct dentry`
source : include/linux/dcache.h

```
struct dentry {
    /* Pointer to the parent directory */
    struct dentry          *d_parent;

    /* Name and inode */
    struct qstr             d_name;
    struct inode           *d_inode;

    /* children and subdirectories */
    struct list_head        d_child;
    struct list_head        d_subdirs;

    /* List of other dentry structures that map to the same
       hash bucket */
    struct hlist_bl_node    d_hash;
}
```

VFS has a generic directory entry structure (`struct dentry`). It is shown in Listing 7.13. One of the key features that is visible is the tree-structured nature of the directory structure. Every directory entry has a parent pointer (`d_parent`). Additionally, each entry has a list of children (`d_child`) and a list of subdirectories (`d_subdirs`). These structures are linked lists. The main aim is to facilitate efficient traversal of the directory structure.

Every directory has a name that is unique to its parent directory (`qstr`). The most important field is the pointer to the inode (`d_inode`) that contains the contents of the directory. Recall that a directory basically stores a table. The data can either be stored as a table, or be a *conceptual table*, where some other data structure is used to represent it. Nevertheless, from the point of view of an outsider a table is stored. Each row of the table corresponds to a file stored in the directory. It can either represent a regular file, directory, device, network socket, etc. Each file has a name, which serves as its unique identified within the directory. The other columns in a row store its metadata, which could be the corresponding file's size and access permissions. A row contains a pointer to the inode of the file.

We can thus summarize our argument as follows. A directory conceptually represents a table, where each row is identified by the file name. It serves as the *key*. The value stores a few metadata fields and a pointer to the file's inode. This file could be a subdirectory that needs to be traversed using the same recursive algorithm.

It is natural to ask what exactly is stored within the disk blocks that contain a directory's contents. VFS does not define the internal structure of the directory. It does not specify how the directory information should be organized within its constituent disk blocks. VFS simply creates a `dentry` structure that stores metadata information and contains a pointer to the inode. It is the role of the specific file system to create the internal structure of the directory.

> **Point 7.6.3**
>
> VFS maintains a cache of inodes and `dentry` structures. For frequently visited directories, there is no need to make a call to the underlying file system and traverse its directory tree. The `dentry` corresponding to the directory can directly be retrieved from the cache (if it is present).

Address Space

Listing 7.14: `struct address_space`
source : `include/linux/fs.h`

```
struct address_space {
    struct inode    *host;              /* host inode */
    struct xarray   i_pages;            /* Pointers to cached
        pages (radix tree) */

    struct rb_root_cached  i_mmap;   /* mapped vmas */
    unsigned long          nrpages;

    /* Functions to bring in and evict folios */
    const struct address_space_operations *a_ops;

    /* Private data to be used by the owner */
    struct list_head    private_list;
    void               *private_data;
}
```

Let us now discuss the page cache. This is a very useful data structure especially for file-backed pages. I/O operations are slow; hence, it should not be necessary to access the I/O devices all the time. It is a far wiser idea to maintain an in-memory page cache that can service reads and writes quickly. A problem of consistency is sadly created. If the system is powered off, then there is a risk of updates getting lost. Thankfully, in modern systems, this behavior can be controlled and regulated to a large extent. It is possible to specify policies. For example, we can specify that when a file is closed, all of its cached data needs to be written back immediately. The `close` operation will be deemed to be successful only after an acknowledgement is received indicating that all the modified data has been successfully written back. There are other methods as well. Linux supports explicit `sync` (synchronization) calls, kernel daemons that periodically sync data to the underlying disk, and write-back operations triggered when the memory pressure increases.

`struct address_space` is an important part of the page cache (refer to Listing 7.14). It stores a mapping (`i_pages`) from an inode to its cached memory

pages (stored as a radix tree). The second map is a mapping `i_mmap` from the inode to a list of `vma` s (stored as a red-black tree). The need to maintain all the virtual memory regions (`vma` s) that have cached pages arises from the fact that there is a need to quickly check if a given virtual address is cached or not. It additionally contains a list of pointers to functions that implement regular operations such as reading or writing pages (`struct address_space_operations`). Finally, each `address_space` stores some private data, which is used by the functions that work on it.

Point 7.6.4

This is a standard pattern that we have been observing for a while now. Whenever we want to define a high-level base class in C, there is a need to create an auxiliary structure with function pointers. These pointers are assigned to real functions by (conceptually) derived classes. In an object-oriented language, there would have been no reason to do so. We could have simply defined a virtual base class and then derived classes could have overridden its functions. However, in the case of the kernel, which is written in C, the same functionality needs to be created using a dedicated structure that stores function pointers. The pointers are assigned to different sets of functions based on the derived class. In this case, the derived class is the actual file system. Ext4 will assign them to functions that are specific to it, and other file systems such as exFat or ReiserFS will do the same.

The role of `struct vma` s needs to be further clarified. A file can be mapped to the address spaces of multiple processes. For each process, we will have a separate `vma` region. Recall that a `vma` region is process-specific. The key problem is to map a `vma` region to a contiguous region of a file. For example, if the `vma` region's start and end addresses are A and B (resp.), we need some record of the fact that the starting address corresponds to the file offset P and the ending address corresponds to file offset Q (note: $B - A = Q - P$). Each `vma` structure stores two fields that help us maintain this information. The first is `vm_file` (a pointer to the file) and the second is `vm_pgoff`. It is the offset within the file – it corresponds to the starting address of the `vma` region. The page offset within the file can be calculated from the address X using the following equation.

$$\texttt{offset} = \frac{(X - \texttt{vm\_start})}{\texttt{PAGE\_SIZE}} + \texttt{vm\_pgoff} \tag{7.3}$$

Here PAGE_SIZE is 4 KB and vm_start is the starting address of the `vma`. Finally, note that we can reverse map file blocks using this data structure as well.

7.6.6 Ext4 File System

Let us now look at the ext4 file system, which is the most popular file system on Linux and Android machines as of 2024. It has two variants. The first variant is very similar to the classical Unix File System. The second variant is designed for large files.

Mapping based on Indirect Blocks
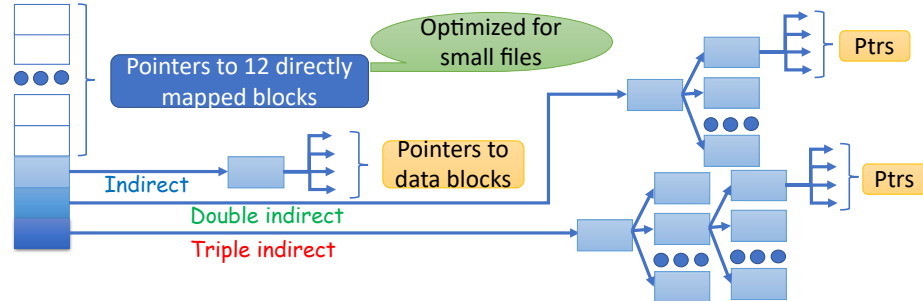
Figure 7.29: Structure of an ext4 inode

Figure 7.29 shows a graphical representation of an `ext4_inode` structure, which is the in-house inode structure of the ext4 file system. Along with some additional metadata fields, it has a data structure for mapping logical blocks to physical blocks. A block is typically 4 KB in the ext4 file system.

The design of the mapping structure is as follows. It has an array of 12 entries that map the first 12 blocks to their physical locations. This is a fast way of accessing the mappings for the first 12 blocks, which makes this a very fast solution for small files. Let us now consider the case of files that have more than 12 blocks. The 13th entry in the array points to a block that contains a set of pointers to data blocks. Such a block is known as an *indirect block*. Assume a block can store κ pointers. This means that the pointers of the blocks numbered $13 \ldots (12 + \kappa)$ can be stored in the indirect block. This means that for slightly larger files, the access time is higher because the indirect block needs to be accessed. What if we need more than $12 + \kappa$ blocks?

In this case, we use double indirect blocks (stored in the 14th entry). Each high-level block points to κ low-level blocks. Furthermore, each low-level block points to κ file blocks. This structure can thus store κ^2 mappings. Similarly, the 15th entry is a triple indirect block. It can point to κ^3 mappings. There is no 16th entry. It is clear that as the file size increases, the average access time also increases because of the indirect blocks. This is a reasonable trade-off.

> **Trivia 7.6.1**
>
> The maximum file size is limited to $12 + \kappa + \kappa^2 + \kappa^3$ blocks. If each block pointer is 32 bits (4 bytes), a block can store 1024 block pointers. Hence, the total file size is limited to $12 + 1024 + 1024^2 + 1024^3$ blocks, which is roughly a billion blocks. Given that each block is 4 KB, the maximum file size is roughly 4 TB.

Mapping based on Extents

The basic idea is similar to the concept of folios – long contiguous sequences of pages in physical and virtual memory. In this case, we define an *extent* to be a contiguous region of addresses on a storage device. Such a region can be fairly

large. Its size can vary from 4 KB to 128 MB. The advantage of large contiguous chunks is that there is no need to repeatedly query a mapping structure for addresses that lie within it. Furthermore, allocation and deallocation is easy. A large region can be allocated in one go. The flip side is that we may end up creating holes as was the case with the base-limit scheme in memory allocation (see Section 6.1.1). In this case, holes don't pose a big issue because extents can be of variable sizes. We can always cover up holes with extents of different sizes. However, the key idea is that we wish to allocate large chunks of data as extents, and simultaneously try to reduce the number of extents. This reduces the amount of metadata required to save information related to extents.
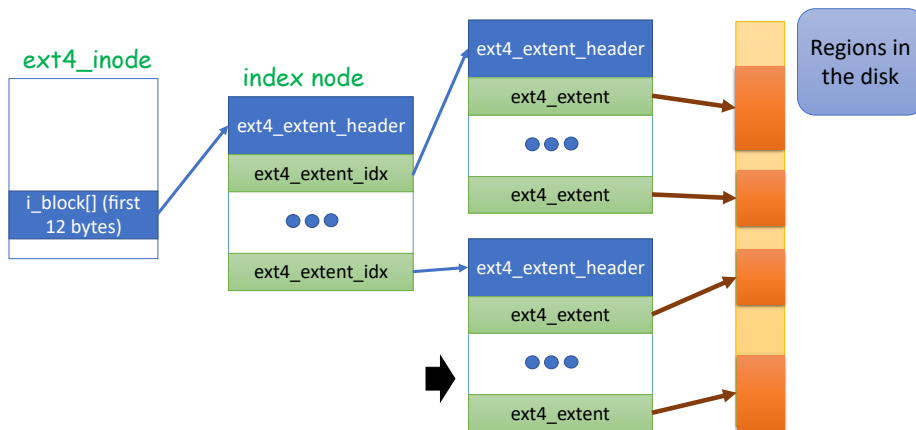
Figure 7.30: Structure of an ext4 inode with extents

The organization of extents is shown in Figure 7.30. In this case, the structure of the **ext4_inode** is different. It can store up to four extents. Each extent points to a contiguous region on the disk. However, if there are more than 5 extents, then there is a need to organize them as a tree (as shown in Figure 7.30). The tree can at the most have 5 levels. Let us elaborate.

There is no need to define a separate ext4 inode for the extent-based filesystem. The **ext4_inode** defines 15 block pointers: 12 for direct block pointers, 1 for the single-indirect block, 1 for the double-indirect block and 1 for the triple-indirect block. Each such pointer is 4 bytes long. Hence, the total storage required in the **ext4_inode** structure is 60 bytes.

The great idea here is to repurpose these 60 bytes to store information related to extents. There is no need to define a separate data structure. The first 12 bytes are used to store the extent header (**struct ext4_extent_header**). The structure is directly stored in these 12 bytes (not its pointer). An ext4 header stores important information about the extent tree: number of entries, the depth of the tree, etc. If the depth is zero, then there is no extent tree. We just use the remaining 48 (60-12) bytes to directly store extents (**struct ext4_extent**). Here also the structures are directly stored, not their pointers. Each **ext4_extent** requires 12 bytes. We can thus store four extents in this case.

The code of an **ext4_extent** is shown in Listing 7.15. It maps a set of contiguous logical blocks (within a file) to contiguous physical blocks (on the

disk). The structure stores the first logical block, the number of blocks and the 48-bit address of the starting physical block. We store the 48 bits using two fields: one 16-bit field and one 32-bit field. An extent basically maps a set of contiguous logical blocks to the same number of contiguous physical blocks. The size of an extent is naturally limited to 2^{15} (32k) blocks. If each block is 4 KB, then an extent can map $32k \times 4$ KB = 128 MB.

Listing 7.15: `struct ext4_extent`
source : fs/ext4/ext4_extents.h

```
struct ext4_extent {
    __le32  ee_block;    /* first logical block */
    __le16  ee_len;      /* number of blocks */
    __le16  ee_start_hi;   /* high 16 bits (phy. block) */
    __le32  ee_start_lo;   /* low 32 bits  (phy. block) */
};
```

Now, consider the case when we need to store more than 4 extents. In this case, there is a need to create an extent tree. Each internal node in the extent tree is represented by the structure `struct ext4_extent_idx` (extent index). It stores the starting logical block number and pointer to the physical block number of the next level of the tree. The next level of the tree is a block (typically 4 KBs). Out of the 4096 bytes, 12 bytes are required for the extent header and 4 bytes for storing some more metadata at the end of the block. This leaves us with 4080 bytes, which can be used to store 340 12-byte data structures. These could either be extents or extent index structures. We are thus creating a 340-ary tree, which is massive. Now, note that we can at the most have a 5-level tree. The maximum file size is thus extremely large. Many file systems limit it to 16 TB. Let us compute the maximum size of the entire file system. The total number of addressable physical blocks is 2^{48}. If each block is 4 KB, then the maximum file system size (known as volume size) is 2^{60} bytes, which is 1 EB (exabyte). We can thus quickly conclude that an extent-based file system is far more scalable than an indirect block-based file system.

Directory Structure

As discussed earlier, it is the job of the ext4 file system to define the internal structure of the directory entries. VFS simply stores structures to implement the external interface.

Listing 7.16: Ext4 directory entry
source : fs/ext4/ext4.h

```
struct ext4_dir_entry_2 {
    __le32  inode;           /* Inode number */
    __le16  rec_len;         /* Directory entry length */
    __u8    name_len;        /* Name length */
    __u8    file_type;       /* File type */
    char    name[EXT4_NAME_LEN];    /* File name */
};
```

Listing 7.16 shows the structure of a directory entry in the ext4 file system. The name of the structure is `ext4_dir_entry_2`. It stores the inode number, length of the directory entry, length of the name of the file, the type of the file

and the name of the file. It basically establishes a connection between the file name and the inode number. In this context, the most important operation is a *lookup* operation. The input is the name of a file, and the output is a pointer to the inode (or alternatively its unique number). This is a straightforward search problem in the directory. We need to design an appropriate data structure for storing the directory entries (e.g.: `ext4_dir_entry_2` in the case of ext4). Let us start with looking at some naive solutions. Trivia 7.6.2 discusses the space of possible solutions.

Trivia 7.6.2

- We can simply store the entries in an unsorted linear list. This will require roughly $n/2$ time comparisons on an average, where n is the total number of files stored in the directory. This is clearly slow and not scalable.

- The next solution is a sorted list that requires $O(log(n))$ comparisons. This is a great data structure if files are not being added or removed. However, if the contents of a directory change, then we need to continuously re-sort the list, which is seldom feasible.

- A hash table has roughly $O(1)$ search complexity. It does not require continuous maintenance. However, it also has scalability problems. There could be a high degree of aliasing (multiple keys map to the same bucket). This will require constant hash table resizing.

- Traditionally, red-black trees and B-trees have been used to solve such problems. They scale well with the number of files in a directory.

In practice, there is a need to create a hybrid data structure. If all the directory entries fit within a block, they are stored as a linear list. A simple linear search is all that is required. However, if more than a block is needed, then ext4 uses a novel data structure: *a hash tree*. It uses the hash of the file's name as the key. The key is used to traverse a B+ tree (see Section C.3.3 in Appendix C) that is limited to three levels. The output is the value, which in this case is a directory entry. We can think of a hash tree as a hybrid of a regular hash table and a B+ tree.

Let us consider the implementation of the hash tree. The first data block stores a `dx_root` structure that stores some metadata. It has a pointer to an array of `dx_entry` data structures. There can be 28 such `dx_entry`s, where each structure maintains a pointer to a file block. These blocks can contain other `dx_entry` structures or directory entries. Note that directory entries can be stored only at the leaf level. This operates as a regular B+ tree. The only caveat is the number of intermediate levels (comprising `dx_entry`s) is limited to 3.

The advantage of such a hash tree is that it is optimized for directories with few files and is also scalable at the same time. It can store directory entries for a large number of files in a convenient tree-shaped structure. This B+ tree allows fast logarithmic-time lookups and has minimal maintenance overheads.

Recall from our discussion on B+ trees that there is often a need to rebalance the tree by splitting nodes and moving keys between them. If keys are deleted, then there is a need to merge nodes. This requires time and is a source of overheads. The process of removing entries is seldom on the critical path. This is because an entry can just be marked as removed and data structure updates can be scheduled for a later time. However, adding a new entry is on the critical path because it needs some space to reside. Hence, from an engineering perspective, it makes some sense to keep some nodes empty, and also have some space empty within each node. If there is a sudden surge in the number of files in a directory, it is possible to quickly allocate directory entries for them using this mechanism.

7.6.7   The exFAT File System

Let us now move on to discussing a few more file systems. The FAT (File Allocation Table) file system used to be quite popular in the 90s. exFAT (extensible FAT) was introduced by Microsoft in 2006. As compared to FAT, it supports larger file sizes. Other than minor modifications, most of the design is the same. As of 2024, the exFAT file system is used in DVDs, USB drives and many embedded systems. The current Linux kernel has extensive support for the exFAT file system.
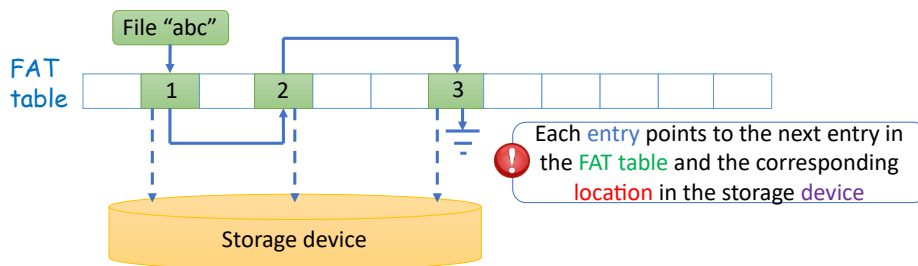
Figure 7.31: Mechanism of storing a file in the exFAT file system

The basic concept is quite simple. We have a long table of entries (the FAT table). This is the primary data structure in the overall design. Each entry has two pointers: a pointer to the next entry in the FAT table (can be null) and a pointer to a *cluster* stored on the disk. A cluster is defined as a set of sectors (on the disk). It is the smallest unit of storage in this file system. We can think of a file as a linked list of entries in the FAT table, where each entry additionally points to a cluster on the disk (or some storage device). Let us elaborate.

Regular Files

Consider a file "abc". It is stored in the FAT file system (refer to Figure 7.31). Let us assume that the size of the file is three clusters. We can number the clusters 1, 2 and 3, respectively. The 1st cluster is the first cluster of the file as shown in the figure. The first FAT table entry of the file in the FAT table has a pointer to this cluster. Note that this pointer is a disk address. Given that this entry is a part of a linked list, it contains a pointer to the next entry (2nd

entry). This entry is designed similarly. It has a pointer to the second cluster of the file. Along with it, it also points to the next node on the linked list (3rd entry). Entry number 3 is the last element on the linked list. Its next pointer is *null*. It contains a pointer to the third cluster.

The structure is thus quite simple and straightforward. The FAT table just stores a lot of linked lists. Each linked list corresponds to a file. In this case a *file* represents both a regular file and a directory. A directory is also represented as a regular file, where the data blocks have a special format.

Almost everybody would agree that the FAT table distinguishes itself on the basis of its simplicity. All that we need to do is divide the total storage space into a set of clusters. We can maintain a bitmap for all the clusters, where the bit corresponding to a cluster is 1 if the cluster is free, otherwise it is busy. Any regular file or directory is a sequence of clusters and thus can easily be represented by a linked list.

Even though the idea seems quite appealing, linked lists have their share of problems. They do not allow random access. This means that given a logical address of a file block, we cannot find its physical address in O(1) time. There is a need to traverse the linked list, which requires $O(N)$ time. Recall that the ext4 file system allowed us to quickly find the physical address of a file block regardless of its design in O(1) time (indirect blocks or extents). This is something that we sacrifice with a FAT table. If we have pure sequential accesses, then this limitation does not pose a major problem.

Directories
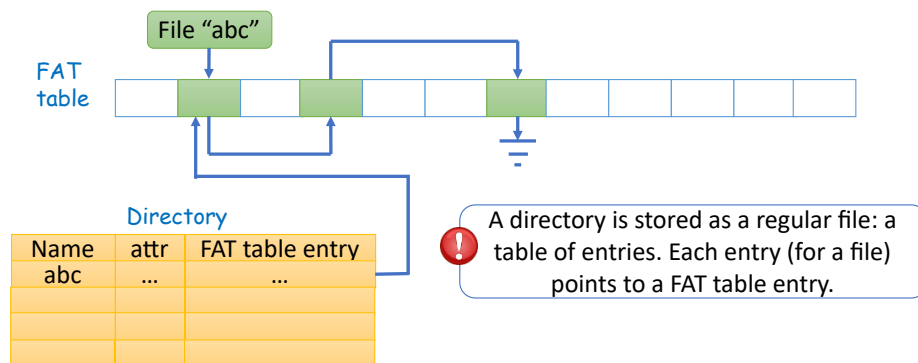
Figure 7.32: Storing files and directories in the FAT file system

Both ext4 and exFAT treat a directory as a regular file to a large extent. It is just a collection of blocks (clusters in the case of exFAT). The "data" associated with a directory has a special format. As shown in Figure 7.32, a directory is a table with several columns. The first column is the file name, which is a unique identifier of the file. Modern file systems such as exFAT support long file names. Sometimes comparing such large file names can be time-consuming. In the interest of efficiency, it is a better idea to hash a file name to a 32 or 64-bit number. Locating a file thus involves simple 32 or 64-bit hash comparison, which is an efficient solution.

The next set of columns store the file's attributes that include a file's status (read-only, hidden, etc.), file length and creation/modification times. The last column is a pointer to the first entry in the FAT table. This part is crucial. It ties a directory entry to the starting cluster of a file via the FAT table. The directory entry does not point to the cluster directly. Instead, it points to the first entry of the file in the FAT table. This entry has two pointers: one points to the first cluster of the file and the other points to the next entry of the linked list.

Due to the simplicity of such file systems, they have found wide use in portable storage media and embedded devices.

7.6.8 Journaling File Systems

Computer systems can crash during the middle of a file write operation. We are looking at a situation where some blocks have been written to permanent storage and the contents of the rest of the blocks are lost. After a system restart, the file will thus be in an inconsistent state. It will be partially written and consequently unusable. Given that crashes can happen at any point of time, it is important to ensure that file systems are robust and resilient to such failures. Otherwise, a lot of important data will get lost. Imagine something like this happening to your bank account or your PhD thesis.

A popular mechanism for dealing with such issues and avoiding file system corruption is *journaling*. A write operation is divided into multiple phases. ① There is a *pre-write* phase where the details all the write requests are written to a journal first. A journal is simply a log of writes that is stored on a storage device. Writing to a journal means that the details of the write access that include the physical address of the location and the blocks' contents are written to the journal's location in stable (durable) storage. After a journal entry has been written, the data cannot be erased even after a power failure. For example, if the journal is stored on a hard disk. Even if there is a power failure, the entries will still be retained. ② Then there is the *commit* phase, where the write is actually effected. The OS sends the write to the storage device and completes it. During this process the system could crash. When the system restarts, it will find unwritten journal entries. The writes can be completed at this stage. This means that the system will attempt to complete the entire write operation once again. Given that write accesses are *idempotent* (the same data can be written to the same location over and over again), there is no problem. ③ Finally, there is a *cleanup* operation, where the journal entry is marked as completed and queued for removal. It is important to note that all updates to the journal are made in stable storage.

Instead of performing long and complicated file system checks, just analyzing the state of journals is enough to discover the integrity of the file system. If there is a problem with some file's contents, its journal entries can just be *replayed*. Table 7.4 outlines the actions that need to be taken when the system crashes. Each row corresponds to a different phase of the write operation.

Assume that the system crashes in the pre-write phase. This can be detected from its journal entry. The journal entry would be incomplete. We assume that it is possible to find out whether a journal entry is fully written to the journal or not. This is possible using a dedicated footer section at the end of the entry. Additionally, we can have an error checking code to verify the integrity of the

Phase	Action
Pre-write	Discard the journal entry
Write	Replay the journal entry
Cleanup	Finish the cleanup process

Table 7.4: Actions that are taken when there the system crashes in different phases of a write operation

entry. In case, the entry is not fully written, then it can simply be discarded.

If the journal entry is fully written, then the next stage commences where a set of blocks on the storage device are written to. This is typically the most time-consuming process. At the end of the write operation, the file system driver updates the journal entry to indicate that the write operation is over. Now assume that the system crashes before this update is made. After a restart, this fact can easily be discovered. The journal entry will be completely written, but there will no record of the fact that the write operation has been fully completed. The entire write operation can be re-done (replayed). Given the idempotence of writes, there are no correctness issues.

Finally, assume that the write operation is fully done but before cleaning up the journal, the system crashes. When the system restarts it can clearly observe that the write operation has been completed, yet the journal entry is still there. It is easy to finish the remaining bookkeeping and mark the entry for removal. Either it can be removed immediately or it can be removed later by a dedicated kernel thread.

7.6.9   Accessing Files in Linux

Example 7.6.1 shows an example of a C program for copying a file in Linux. The aim is to copy the contents of "a.txt" to a new file "b.txt". We start with making the `fopen` call that opens the file. This library call makes the `open` function call. Subsequently, VFS locates the file's inode and returns a handle to it.

Linux maintains a table of open files for each process and also a systemwide open file table. Whenever a new file is opened, its details are added to the systemwide open file table, if a corresponding entry does not exist. Next, information regarding the recently opened file is added to the per-process open file table with a reference to the entry in the systemwide open file table. The index in this per-process open file table is the integer *file descriptor*. It is used to uniquely identify an open file (within the process). There are some standard file descriptors in Linux that are pre-defined. 0 standards for the standard input from the shell (`stdin`). 1 is the standard output stream (`stdout`) and 2 is the standard error stream (`stderr`). It is important to note that two processes may actually point to two different files even if their file descriptors have the same values. This is because the contents of their per-process open file tables are different.

Along with the integer file descriptor, Linux maintains buffers to store data that is read or written to the file. It also stores the status of the file (opened for reading, writing or appending), the current file pointer and information related

to errors encountered. All of this information is bundled in the FILE structure that is returned by the `fopen` library call. Note that this call specifically returns a pointer to a FILE structure because it is always more efficient to do so. Insofar as the C program is concerned, the FILE pointer is the only handle to the open file that it has. If it is NULL after an `fopen` call, then it means that for various reasons the file could not be opened. Either it does not exist or the user does not have adequate permissions.

Example 7.6.1

Write a program in C to copy a file. Copy the contents of "a.txt" to another file "b.txt".

Answer:

Listing 7.17: Copying a file

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    char c;
    FILE *src_file, *dst_file;

    /* Open the source and destination files */
    src_file = fopen("a.txt", "r");
    dst_file = fopen("b.txt", "w");

    if (src_file == NULL) {
        printf("Could not open a.txt \n");
        exit(1);
    }

    if (dst_file == NULL) {
        fclose(src_file);
        printf("Could not open b.txt \n");
        exit(1);
    }

    /* Iteratively transfer bytes */
    while ((c = fgetc(src_file)) != EOF) {
        fputc(c, dst_file);
    }

    /* Close the files */
    fclose(src_file);
    fclose(dst_file);

    printf("Successfully copied the file \n");
}
```

On similar lines, we open the file "b.txt" for writing. In this case, the mode is "w", which means that we wish to write to the file. The corresponding mode

for opening the source file ("a.txt") was "r" because we opened it in read-only mode. Subsequently, we keep reading the source file character by character and keep writing them to the destination file. If the character read is equal to EOF (end of file), then it means that the end of the file has been reached and there are no more valid characters left. The C library call to read characters is fgetc and the library call to write a character is fputc. It is important to note that both these library calls take the FILE handle (structure) as the sole argument for identifying the file that has been opened in the past. Here, it is important to note that a file cannot be accessed without opening it first. This is because opening a file creates some state in the kernel that is subsequently required while accessing it. We are already aware of the changes that are made such as adding a new entry to the systemwide open file table, per-process open file table, etc.

Finally, we close both the files using the fclose library calls. They clean up the state in the kernel. They remove the corresponding entries from the per-process file table. The entries from the systemwide table are removed only if there is no other process that has simultaneously opened these files. Otherwise, we retain the entries in the systemwide open file table.

Let us consider the next example (Example 7.6.2) that opens a file, maps it to memory and counts the number of 'a's in the file. We proceed similarly. We open the file "a.txt", and assign it to a file handle file. In this case, we need to also retrieve the integer file descriptor because there are many calls that need it. This is easily achieved using the fileno function.

Example 7.6.2

Open a file "a.txt", and count the number of 'a's in the file.

Answer:

Listing 7.18: Count the number 'a's in a file

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <sys/mman.h>

int main() {
    FILE *file;
    int fd;
    char *buf;
    struct stat info;
    int i, size, count = 0;

    /* Open the file */
    file = fopen("a.txt", "r");
    if (file == NULL) {
        printf ("Error opening file a.txt");
        exit(1);
    }
    fd = fileno(file);
```

```
    /* Get the size of the file */
    fstat(fd, &info);
    size = info.st_size;

    /* Memory map the file */
    buf = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd,
        0);

    /* Count the number of 'a's */
    for (i = 0; i < size; i++) {
        if (buf[i] == 'a')
            count++;
    }

    /* Unmap and close the file */
    munmap(buf, size);
    fclose(file);

    printf("The number of 'a' chars is %d\n", count);
}
```

Subsequently, we invoke the `fstat` function to get the statistics associated with a file. It returns all that information in the `info` structure. Next, we map the open file to memory. The `mmap` function takes a bunch of arguments. The first argument is the address at which the file to be mapped. It is optional and in this case we are specifying it to be NULL, which means that the OS can decide where to place the contents of the file. In the latter case, the return value of the `mmap` call is the virtual address at which the file was mapped. The next argument is the size of the file, which we get from the `info` structure (output of `fstat`). The next argument is the type of permission that is required. In this case, we just want read permission (`PROT_READ`). It is possible to share the mapped the region with other processes by specifying the subsequent argument as `MAP_SHARED`. There is no such requirement in this case. Hence, we pass the argument `MAP_PRIVATE`. The last two arguments are the number of the file descriptor and the starting offset of the region within the file, respectively. If this process is successful, then a part of the process's address space is memory-mapped. Internally, the magic happens in the page table of the process. A part of the virtual address is mapped to physical pages in the page cache. These pages store file blocks in physical memory. This means that any change made in the process reflects in the contents of the physical pages that are a part of the page cache. Depending upon the policy, the page cache writes the data in its pages to the file stored on disk. This process is not visible to the user process. Insofar as it is concerned, it simply directs the writes to the memory-mapped region in its virtual address space. The rest of the data transport from a program's virtual address space to the page cache's pages to the file stored on the disk happens automatically !!!

In our example, we access a file as if it is an array stored in memory. The statement `buf[i] == 'a'` does exactly that. Finally, we unmap the memory region and close the file.

7.6.10 Pipes

Let us now look at a special kind of file known as a *pipe*. A pipe functions as a producer-consumer queue. Even though modern pipes have support for multiple producers and consumers, a typical pipe has a process that writes data at one end, and another process that reads data from the other end. There is built-in synchronization. This is a fairly convenient method of transferring data across processes. There are two kinds of pipes: named and anonymous. We shall look at anonymous pipes first.

Anonymous Pipes

An anonymous pipe is a pair of file descriptors. One file descriptor is used to write, and the other is used to read. This means that the writing process has one file descriptor, which it uses to write to the pipe. The reading process has one more file descriptor, which it uses to read. A pipe is a *buffered* channel, which means that if the reader is inactive, the pipe buffers the data that has not been read. Once the data is read, it is removed from the pipe. Example 7.6.3 shows an example.

Example 7.6.3

Write a program that uses anonymous pipes.

Answer:

Listing 7.19: Using an anonymous pipe across a parent-child process pair

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main() {
    pid_t pid;
    int pipefd[2];
    char msg_sent[] = "I love my OS book";
    char msg_rcvd[30];

    /* Create the pipe (file descriptor pair) */
    /* 0 is the read end and 1 is the write end */
    pipe(pipefd);

    /* fork */
    pid = fork();

    if (pid > 0) {
    /* parent process */
        close(pipefd[0]);

    /* write the message */
        write(pipefd[1], msg_sent, strlen(msg_sent) +
            1);
        close(pipefd[1]);
    } else {
```

```
    /* Child process */
        close(pipefd[1]);

    /* read the message */
        read(pipefd[0], msg_rcvd, sizeof(msg_rcvd));
        close(pipefd[0]);

    /* print the message */
        printf("Message received: %s\n", msg_rcvd);
    }
}
```

As we can see in the example, the `pipe` library call (and system call) creates a pair of file descriptors. It returns a 2-element array of file descriptors. 0 is the read end, and 1 is the write end. In the example, the array of file descriptors is passed to both the parent and the child process. Given that the parent needs to write data, it closes the read end (`pipefd[0]`). Note that instead of using `fclose`, we use `close` that takes a file descriptor as input. In general, the library calls with a prefix of 'f' are at a high level and have lower flexibility. On the other hand, calls such as `open`, `close`, `read` and `write` directly wrap the corresponding system calls and are at a much lower level.

The parent process quickly closes the file descriptor that it does not need (read end). It writes a string `msg_sent` to the pipe. The child process is the reader. It does something similar – it closes the write end. It reads the message from the pipe, and then prints it.

Named Pipes
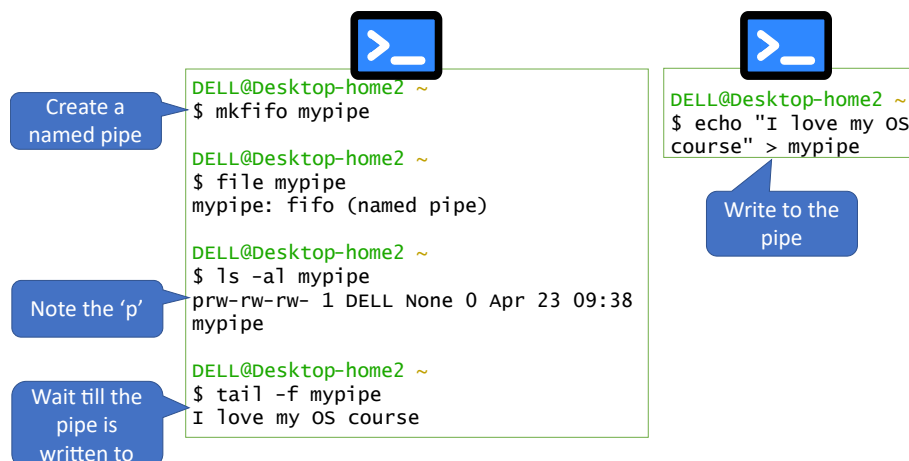
Figure 7.33: Pipes in Linux

Figure 7.33 shows a method for using named pipes. In this case the `mkfifo` command is used to create a pipe file called `mypipe`. Its details can be listed

with the `file` command. The output shows that it is a named pipe, which is akin to a producer-consumer FIFO queue. A directory listing shows the file to be of type 'p'. Given that the file `mypipe` is now a valid file in the file system, a process running on a different shell can simply write to it. In this case, we are writing the string "I love my OS course" to the pipe by redirecting the output stream to the pipe. The '>' symbol redirects the output to the pipe. The other reading process can now read the message from the pipe by using the `tail` shell command. We see the same message being printed.

Using such named pipes gives processes a convenient mechanism to pass messages between each other. They do not have to create a new pipe all the time. One end of the pipe can just be treated as a regular file that is being written to. As we have seen the '>' symbol redirects the standard output stream to the pipe. Similarly, the other side, which is the read end can be used by any program to read any messages present in the pipe. Here also it is possible to redirect the standard input to a file using the '<' symbol.

7.7 Summary and Further Reading

7.7.1 Summary

7.7.2 Further Reading

Exercises

Ex. 1 — When is a source synchronous bus used?

Ex. 2 — Why are modern buses like USB designed as serial buses?

Ex. 3 — What is the advantage of RAID 5 over RAID 4?

Ex. 4 — Give an example where RAID 3 (striping at the byte level) is the preferred approach.

Ex. 5 — What is the advantage of a storage device that rotates with a constant linear velocity?

** **Ex. 6 —** RAID 0 stripes data – stores odd numbered blocks in disk 0 and even numbered blocks in disk 1. RAID 1 creates a mirror image of the data (disk 0 and disk 1 have the same contents). Consider RAID 10 (first mirror and then stripe), and RAID 01 (first stripe and then mirror). Both the configurations will have four hard disks divided into groups of two disks. Each group is called a first-level RAID group. We are essentially making a second-level RAID group out of two first-level RAID groups. Now, answer the following questions:

a)Does RAID 01 offer the same performance as RAID 10?

b)What about their reliability? Is it the same? You need to make an implicit assumption here, which is that it is highly unlikely that both the disks belonging to the same first-level RAID group will fail simultaneously.

Ex. 7 — The motor in hard disks rotates at a constant angular velocity. What problems does this cause? How should they be solved?

Ex. 8 — We often use bit vectors to store the list of free blocks in file systems. Can we optimize the bit vectors and reduce the amount of storage?

Ex. 9 — What is the difference between the contents of a directory, and the contents of a file?

Ex. 10 — Describe the advantages and disadvantages of memory-mapped I/O and port-mapped I/O.

Ex. 11 — Give an example of a situation in which ordinary pipes are more suitable than named pipes, and an example of a situation in which named pipes are more suitable than ordinary pipes. Explain your answer.

Ex. 12 — Give an example of a situation in which sockets are more suitable than shared memory and an example of a situation in which shared memory is more suitable than sockets for inter process communication. Explain your answer.

Ex. 13 — Consider a hard disk that rotates at 15,000 rotations per minute (RPM) and has a transfer rate of 50 Megabytes/sec. The seek time is 5 ms. How much time does it take to transfer a 512 byte sector (on an average)?

Ex. 14 — How does memory-mapped I/O work in the case of hard disks? We need to perform reads, writes and check the status of the disk. How does the processor know that a given address is actually an I/O address, and how is this communicated to software? Are these operations synchronous or asynchronous? What is the advantage of this method over a design that uses regular I/O ports? Explain your answers.

Ex. 15 — Explain the working of the FAT file system.

Ex. 16 — FAT file systems find it hard to support `seek` operations. How can a FAT file system be modified to support such operations more efficiently?

Ex. 17 — What are the advantages of representing everything as a file in Linux?

Ex. 18 — How is the file system for a flash device different from that of a file system tailored for hard disks?

* **Ex. 19** — Assume a storage system with a three-layered structure: small amount of volatile RAM memory (fast), numerous flash (SSD) disks (medium speed), and an array of hard disks (very slow). We want to create a filesystem for a service like Instagram (image sharing). Design a file system with the following features.

a)A user can add any number of images (variable sizes).

b)Each image can be commented upon, and there can be replies to comments as well. However, we cannot comment on replies. The size of each comment and reply is limited to 256 bytes.

Ex. 20 — Most flash devices have a small DRAM cache, which is used to reduce the number of PE-cycles and the degree of read disturbance. Assume that the DRAM cache is managed by software. Suggest a data structure that can be created on the DRAM cache to manage flash reads and writes such that we minimize the #PE-cycles and read disturbance.

* **Ex. 21** — We want to implement a *container* (such as Docker) with a virtual file system. It is a file system that sits on top of the host's file system. It contains a subset of the files in the underlying file system (maintained by the host OS). Any process running in the container can only access the virtual file system. If it decides to modify a file or make some other change to the file system such as deleting a file or changing the metadata, then the changes are confined to the virtual file system. Other user processes running on the host OS do not see these updates. Propose a method to implement such a virtual file system.

Ex. 22 — Answer the following questions with respect to devices and device drivers:

a)Why do we have both software and hardware request queues in `structrequest_queue`?

b)Why do device drivers deliberately delay requests?

c)Why should we just not remove (eject) a USB key?

d)What can be done to ensure that even if a user forcefully removes a USB key, its FAT file system is not corrupted?

Ex. 23 — Do different partitions of a hard disk have separate request queues?

Ex. 24 — What is the purpose of `struct bio`?

Ex. 25 — Suggest an algorithm for periodically draining the page cache (syncing it with the underlying storage device). What happens if the sync frequency is very high or very low?

* **Ex. 26** — A container such as Docker virtualizes the file system. It stores the difference between the "virtual" file system and the underlying host file system (at the level of full files). Let us assume that there are some large files in the host file system that the container writes to. We however do not want to replicate the entire file. We want to replicate only those parts of a large file that have been modified by a Docker process, and we want to make minimal changes to Docker's standard inode based file system. Propose the design of a such a file system.

** **Ex. 27** — Design a file system for a system like Twitter/X. Assume that each tweet (small piece of text) is stored as a small file. The file size is limited to 256 bytes. Given a tweet, a user would like to take a look at the replies to the tweet, which are themselves tweets. Furthermore, it is possible that a tweet

may be retweeted (posted again) many times. The "retweet" (new post) will be visible to a user's friends. Note that there are no circular dependences. It is never the case that: (1) A tweets, (2) B sees it because B is A's friend, (3) B retweets the same message, and (4) A gets to see the retweet. Design a file system that is suitable for this purpose.

Ex. 28 — Consider a large directory in the exFAT file system. Assume that its contents span several blocks. How is the directory (represented as a file) stored in the FAT table? What does each row in a directory's data block look like? How do we create a new file and allocate space to it in this filesystem? For the last part, explain the data structures that we need to maintain. Justify the design.

Virtualization and Security

8.1 Summary and Further Reading

8.1.1 Summary

8.1.2 Further Reading

Exercises

Ex. 1 — Explain the different types of hypervisors.

Ex. 2 — Describe the trap-and-emulate method. How does it work for interrupts, privileged instructions and system calls?

** **Ex. 3** — Most proprietary software use a license server to verify if the user has sufficient credentials to run the software. Think of a "license server" as an external server. The client sends its id, and IP address (cannot be spoofed) along with some more information. After several rounds of communication, the server sends a token that the client can use to run the application only once. The next time we run the application, a fresh token is required. Design a cryptographic protocol that is immune to changing the system time on the client machine,

replay attacks, and man-in-the-middle attacks. Assume that the binary of the program cannot be changed.

** **Ex. 4** — We want to implement containers in an operating system. A container is a mini virtual machine. Here are the features that a container should have:

  a) Every container has its own set of user ids and passwords.

  b) We can launch a process inside a container. It will not be visible to processes in other containers. At any point of time, processes from multiple containers might be running. Note that a container is not an operating system, nor a VMM. It is just a thin layer on top of the OS.

  c) Initially, every process sees the native file system of the underlying OS. However, the moment, we change a file, a fresh copy is created for that container. For example, if we create a new version of `/etc/passwd` in a container, then only that container will see the updated version of this file. This change will not be visible outside the container.

  d) It should be possible to restrict the privileges of processes in a container. For example, they might not be able to access the network, or certain sectors on the disk.

Suggest a mechanism to implement all of these as efficiently as possible.

** **Ex. 5** — Let us design an operating system that supports record and replay. We first run the operating system in record mode, where it executes a host of applications that interact with I/O devices, the hard disk, and the network. A small module inside the operating system records all the events of interest. Let us call this the record phase.

After the record phase terminates, later on, we can run a replay phase. In this case, we shall run the operating system and all the constituent processes exactly the same way as they were running in the record phase. The OS and all the processes will show exactly the same behavior, and also produce exactly the same outputs in the same order. To an outsider both the executions will be indistinguishable. Such systems are typically used for debugging and testing, where it is necessary to exactly reproduce the execution of an entire system.

Your answer should at least address the following points:

  a) What do we do about the time? It is clear that we have to use some notion of a logical time in the replay phase.

  b) How do we deliver I/O messages from the network or hard disk, and interrupts with exactly the same content, and exactly at the same times?

  c) What about non-determinism in the memory system such as TLB misses, and page faults?

  d) How do we handle inherently non-deterministic instructions such as reading the current time and generating a random number?

* **Ex. 6** — We wish to encrypt a part of the physical memory space. How do we modify the virtual memory mechanism to support this? Assume that the OS is trustworthy.

Ex. 7 — How does the VMM keep track of updates to the guest OS's page tables in shadow and nested paging?

Ex. 8 — Answer the following questions with respect to virtual machines:

Ex. 9 — If there is a context switch in the guest OS, how does the VMM get to know the id (or something equivalent) of the new process (one that is being swapped in)? Even if the VMM is not able to find the pid of the new process being run by the guest OS, it should have some information available with it such that it can locate the page table and other bookkeeping information corresponding to the new process.

Ex. 10 — How do we virtualize the TLB?

Ex. 11 — Why does HW-assisted virtualization allow the host's shadow page table and the guest page table to go out of sync, and why is this not a problem?

The X86-64 Assembly Language

In this book, we have concerned ourselves only with the Linux kernel and that too in the context of the x86-64 (64-bit) ISA. This section will thus provide a brief introduction to this ISA. It is not meant to be a definitive reference. For a deeper explanation, please refer to the textbook on basic computer architecture by your author [Sarangi, 2021].

The x86-64 architecture is a logical successor of the x86 32-bit architecture, which succeeded the 16 and 8-bit versions, respectively. It is the default architecture of all Intel and AMD processors as of 2023. The CISC ISA got complicated with the passage of time. From its early 8-bit origins, the development of these processors passed through several milestones. The 16-bit version arrived in 1978, and the 32-bit version arrived along with Intel 80386 that was released in 1985. Intel and AMD introduced the x86-64 ISA in 2003. The ISA has become increasingly complex over the years and hundreds of new instructions have been added henceforth, particularly vector extensions (a single instruction can work on a full vector of data).

A.1   Registers
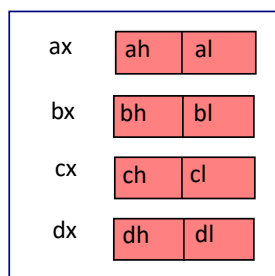
Figure A.1: Two virtual 8-bit registers within a 16-bit register

The x86-64 ISA has 16 registers. These 16 registers have an interesting history. In the 8-bit version, they were named simply a, b, c and d. In the

16-bit avatar of the ISA, these registers were simply extended to 16 bits. Their names changed though, for instance `a` became `ax`, `b` became `bx`, and so on. As shown in Figure A.1, the original 8-bit registers continued to be accessible for backward compatibility. Each 16-bit register was split into a high and low part. The lower 8 MSB bits are addressable using the specifier `al` (low) and bits 9-16 are addressable using the register `ah` (high).

A few more registers are present in the 16-bit ISA. There is a stack pointer `sp` (top of the stack), a frame pointer `bp` (beginning of the activation block for the current function), and two index registers for performing computations in a loop via a single instruction (`si` and `di`). In the 32-bit variant, a prefix 'e' was added. `ax` became `eax`, so on and so forth. Furthermore, in the 64-bit variant the prefix 'e' was replaced with the prefix 'r'. Along with these registers, 8 new registers were added – `r8` to `r15`. This is shown in Figure A.2. Note that even in the 64-bit variant of the ISA, known as x86-64, the 8, 16 and 32-bit registers are accessible. It is just that these registers exist virtually (as a part of larger registers).
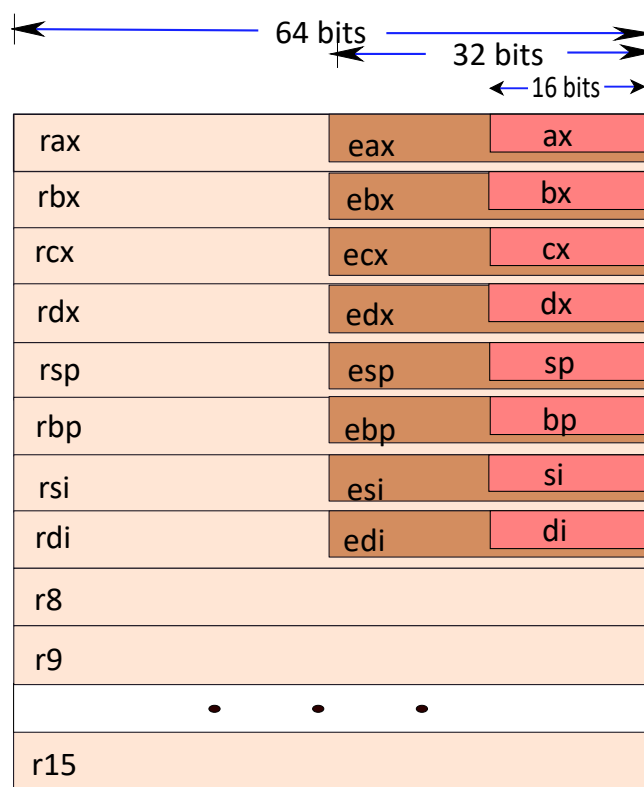
Figure A.2: The registers in the x86-64 ISA

Note that unlike newer RISC ISAs, the program counter is not directly accessible. It is known as the *instruction pointer* in the x86 ISA, which is not visible to the programmer. Along with the program counter, there is also a `flags` register that becomes `rflags` in x86-64. It stores all the ALU flags. For example, it stores the result of the last compare instruction. Subsequent branch

instructions use the result of this compare instruction for deciding the outcome of conditional branches (refer to Figure A.3).
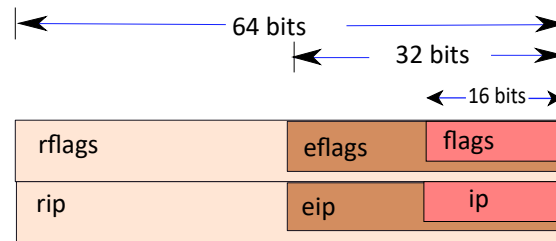
Figure A.3: Special registers in the x86 ISA

There are a couple of fields in the `rflags` register that are commonly used. Each field typically requires 1 bit of storage and has a designated bit position in the 64-bit register `rflags`. If the corresponding bit position is set to 1, then it means that the corresponding flag is set otherwise it is unset (flag is false). `OF` is the integer overflow flag, `CF` is the carry flag (generated in an addition), the `ZF` flag is set when the last comparison resulted in an equality, and the `SF` sign flag is set when the last operation that could set a flag resulted in a negative result. Note that a comparison operation is basically implemented as a subtraction operation. If the two operands are equal, then the comparison results in an equality (zero flag is set) otherwise if the first operand is less than the second operand, then the result is negative and the sign flag is set to 1 (result is negative).

Floating Point Registers

The basic floating point register set has 8 registers (see Figure A.4). They are 80 bits wide. This is known as extended precision (more than double precision, which is 64 bits). The organization of floating point registers is quite interesting. The registers are arranged as a stack. The stack top is `st0` and the bottom of the stack is `st7`. If a new value is pushed to the stack, then the value at the stack top moves to `st1`. The rest of the registers are also pushed back by 1. For example, the old `st3` becomes the new `st4`. Every register is basically a position in this model. Additionally, registers are also directly accessible. For example, we can directly use the register specifier `st5`. However, the connection between the value and the register location `st5` will break the moment there is a `push` or `pop` operation on the stack. Floating point operations are typically structured as operations that use the top of the stack (`st0`) as the implicit operand, and often involve push/pop operations. For example, the floating point add operation adds the first two entries on the stack, pops one entry and replaces the top of the stack with the sum. This reduces the overall code size dramatically. This was an important motivation when the floating point ISA was designed.

The stack-based model is typically adopted by very simple machines that are easy to program. We basically restrict our set of instructions to arithmetic instructions, load/store instructions, push and pop. The 80387 math coprocessor that used to be attached to erstwhile Intel processors to provide floating
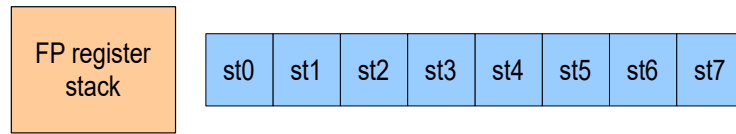
Figure A.4: Floating point registers

point capabilities used this stack-based model. This basic programming model has remained in the x86 ISA. Because backward compatibility is a necessary requirement, this model is still present. With the advent of fast hardware and compiler technology, this has not proved to be a very strong impediment.

A.2 Basic Instructions

There are two formats in which x86 instructions are written. There is an Intel format and there is an AT&T format. In both the formats one of the sources is also the destination operand. In the Intel format, the first operand is both a source and a destination, whereas in the AT&T format, the second operand is both a source and a destination. The AT&T format is the default format that the popular open source compiler `gcc` generates unless it is instructed otherwise. We will thus use the AT&T format in this book.

Examples of some instructions are as follows.

```
movq $3, %rax
movq $4, %rbx
addq %rbx, %rax
movq %rax, 8(%rsp)
```

The basic `mov` operation moves the first operand to the second operand. The first operand is the source and the second operand is the destination in this format. Each instruction admits a suffix (or modifier), which specifies the number of bits that we want it to operate on. The 'q' modifier means that we wish to operate on 64 bits, whereas the 'l' modifier indicates that we wish to operate on 32-bit values. In the instruction `movq $3, %rax`, we move the number 3 (prefixed with a '$') to the register `rax`. Note that all registers are prefixed with a percentage ('%') symbol. Similarly, the next instruction `movq $4, %rbx` moves the number 4 to the register `rbx`. The third instruction `addq %rbx, %rax` adds the contents of register `rbx` to the contents of register `rax`, and stores the result in `rax`. Note that in this case, the second operand `%rax` is both a source and a destination. The final instruction stores the contents of `rax` (that was just computed) to memory. In this case, the memory address is computed by adding the base address that is stored in the stack pointer (`%rsp`) with the offset 8. The `movq` instruction moves data between registers as well as between a register and a memory location. It thus works as both a load and a store. Note that we cannot transfer data from one memory location to another memory location using a single instruction. In other words, it is not possible to have two memory operands in an instruction.

Let us look at the code for computing the factorial of the number 10 in Listing A.1. In this case, we use the 32-bit version of the ISA. Note that it is

perfectly legal to do so in a 64-bit processor for power and performance reasons. In the code shown in Listing A.1, `eax` stores the number that we are currently multiplying and `edx` stores the product. The `imull` instruction multiplies the partial product (initialized to 1) with the index.

Listing A.1: Code for computing factorial(10)

```
    movl    $1, %edx        # prod = 1
    movl    $1, %eax        # i = 1
.L2:
    imull   %eax, %edx      # prod = prod * i
    addl    $1, %eax        # i = i + 1
    cmpl    $11, %eax       # compare i with 11
    jne .L2                 # if (!(i == 11)) goto .L2
```

Format of Memory Operands

The x86 instruction set has elaborate support for memory operands. Since it is a CISC instruction set, it supports a wide variety of addressing modes, particularly for memory operands. The standard format for specifying a memory address in x86 assembly is as follows: `seg:disp(base,index,scale)`. The address is computed as shown in Equation A.1. Here, we are assuming that the `seg` segment register stores a base address that gets added to the computed address. Section 2.2.4 discusses segmentation in x86 in great detail. Often the segment register is not specified. For different types of accesses, default segment registers are used. For example, the code segment register is used for code and the data segment register is used for data.

$$address = seg + (base + index * scale + disp) \qquad (A.1)$$

`base` refers to the base address register. It is additionally possible to specify an index, which is also a register. Its contents are added to the base address (stored in the base register). The index register can optionally be scaled by the value specified in the `scale` parameter. The `scale` parameter is very useful while implementing array accesses. The base address of the array is stored in the `base` register, the array index is stored in the `index` register and `scale` is used to specify the size of the data type. For instance, if the data type is an integer, then the scale is equal to 4. If the data type is a double, then the scale is equal to 8, so on and so forth. It is additionally possible to specify a fixed offset, which is also known as the displacement (`disp` field). This field is particularly important while specifying the address of variables stored on the stack or in some regions where the location is a fixed offset away from the start of the region. The `disp` field can also be specified standalone, when nothing else is specified. The advantage here is that we can implement direct memory addressing, where the memory address is specified directly – it need not be calculated using the base or index registers.

Let us consider a few examples. The address (`4(%esp)`) has `esp` as the base register with 4 as the displacement. In this case, the address is being specified relative to the value of the stack pointer stored in `esp`. Another example of an address is (`%eax,%ebx`). This address does not have a displacement or a scale. It just has a base register (`eax`) and an index register `ebx`. Let us now look

at another address in its full glory, `-32(%eax,%ebx,0x4)`. The displacement is (-32) and the scale is 4 (specified in the hex format).

As we can observe, in x86, the memory operand field is extremely expressive, and it can be used to specify a wide range of operands in a reasonably large number of formats.

Appendix B

Compiling, Linking and Loading

B.1 The Process of Compilation

It is important to understand the process of compiling, linking and loading. Many students often get confused with these concepts and don't understand what they really mean and how they can be used to build large software. Let us first look at the simplest of these steps, which is the process of *compilation*. A compiler's job can be broken into two parts: frontend and backend. The frontend part of the compiler reads a C file, ensures that the syntax is correct, and it is well-formed. If there are no errors, we proceed to the second stage, which involves invoking the backend routines. Of course, if there is an error, the programmer is informed and then the programmer needs to make appropriate changes to the program such that the compilation errors go away.

The frontend basically reads a sequence of bytes (the C file) and converts it into a sequence of *tokens*. This process is known as lexical analysis. The sequence of tokens is then used to create a *parse tree*. Often programs like *yacc* and *bison* are used to specify the grammar associated with a programming language and automatically create a parse tree for a source code file (like a C file). The parse tree contains the details of all the code that is there in the C file. This includes a list of all the global and statically defined variables, their data types, all the functions, their arguments, return values and all the code statements within the functions. In a certain sense, the parse tree is a representation that passes certain syntactic and semantic checks, completely represents the contents of the C file, and is very easy to handle. It incorporates many syntactic details, which are not really required to create machine code. Hence, the parse tree is used to construct a simpler representation that is far easier to process and is devoid of unnecessary details – this simpler tree is known as the *Abstract Syntax Tree* or AST. The AST is the primary data structure that is processed by the backend of the compiler.

B.1.1 Compiler Passes

The backend of the compiler is structured as a sequence of multiple passes. Each pass reads the abstract syntax tree and then produces a representation that is

437

semantically equivalent, yet is a more optimized version. For instance, there could be pieces of code that will never be invoked in any execution. Such pieces of code are known as *dead code*. One pass could be dead code removal where all such pieces of code are identified and removed. Another common compiler pass is an optimization pass called *constant folding*. Consider a statement in a C file, which is as follows: `int a = 5 + 3 + 9;`. In this case, there is no need to actually put the values 5, 3 and 9 in registers and add them. The compiler can directly add 5, 3 and 9 and set the value of variable `a` to 17. Such optimizations save many instructions and make the program quite efficient. Compiler writers are quite ingenious and have proposed tens of optimizations. In fact, the optimization flags -O1, -O2 and -O3 in the popular `gcc` compiler specify the aggressiveness of optimizations. For example, -O1 comprises fewer optimization passes than -O2, so on and so forth. Having more optimization passes increases the chances of producing more efficient code. However, often diminishing returns set in, and sometimes the optimizations also tend to cancel each other's gains. Hence, it can so happen that overly optimizing a program actually leads to a slowdown. Hence, there is a need to understand which passes a program is actually going to benefit from.

The important point to note is that this is a highly complicated process where a pass is designed for a particular kind of optimization and the process of backend compilation is essentially a sequence of multiple passes. Clearly, the nature of the passes matters as well as their sequence. The backend starts with working on ASTs, which gradually get optimized and simplified. At some point, compilers start producing code that looks like machine code. These are known as intermediate representations (IRs). Initially, the intermediate representations are at a reasonably high-level, which means that they are not ready to be converted to assembly code just yet. Gradually, each instruction in the intermediate representation starts getting closer and closer to machine instructions. These are referred to as medium and low level intermediate representations. For obvious reasons, low-level IR cannot be used to perform major optimizations that require a lot of visibility into the overall structure of the code. However, simple instruction-level optimizations can be made easily with IR representations that are close to the final machine code. These are also known as *peephole optimizations*. Gradually, over several compiler passes, the IR starts representing the machine code.

The last step in the backend of the compiler is code generation. The low-level IR is converted to actual machine code. It is important for the compiler to know the exact semantics of instructions on the target machine. Many times there are complex corner cases where we have floating point flags and other rarely used instructions involved. They have their own set of idiosyncrasies. Needless to say, any compiler needs to be aware of them, and it needs to use the appropriate set of instructions such that the code executes as efficiently as possible. We need to guarantee 100% correctness. Furthermore, many compilers as of 2023 allow the user to specify the compilation priorities. For instance, some programmers may be looking at reducing the code size and for them performance may not be that great a priority. Whereas, for other programmers, performance may be the topmost priority. Almost all modern compilers are designed to handle such concerns and generate code accordingly.

B.1.2   Dealing with Multiple C Files

We may be very happy at this stage because a full C file has been fully compiled and has been converted to machine code. However, some amount of pessimism is due because most modern software projects typically consist of thousands of files that have been written by hundreds of developers. As a result, any project will have hundreds or thousands of C files. Now, it is possible that a function in one C file actually calls functions defined in other C files, and there is a complex dependence structure across the files. The same holds true for global variables as well. Hence, we observe that when a C file is being compiled, the addresses of many functions as well as global variables that are being used may not be known.

Point B.1.1

If we have many source files, the addresses of many variables and functions will not be known at the compilation stage. This is because they are defined in other files. Their addresses need to be <u>resolved</u> later.

Object Files

Let us now take a look at Figure B.1. It shows the different phases of the overall compilation process. Let us look at the first phase, which is converting a C file to a .o file. The .o file is also known as an *object file*, which represents the compiler output obtained after compiling a single C file. It contains machine code corresponding to the high-level C code along with other information. It is of course possible that a set of symbols (variables and functions) do not have their addresses set correctly in the .o file because they were not known at the time of compilation. All such symbols are identified and placed in a *relocation table* within the .o file. The linking process or the *linker* is then tasked with taking all the individual .o files and combining them into one large binary file, which can be executed by the user. This binary file has all the symbols' addresses defined (we will relax this assumption later). Note that we shall refer to the final executable as the *program binary* or simply as the *executable*.
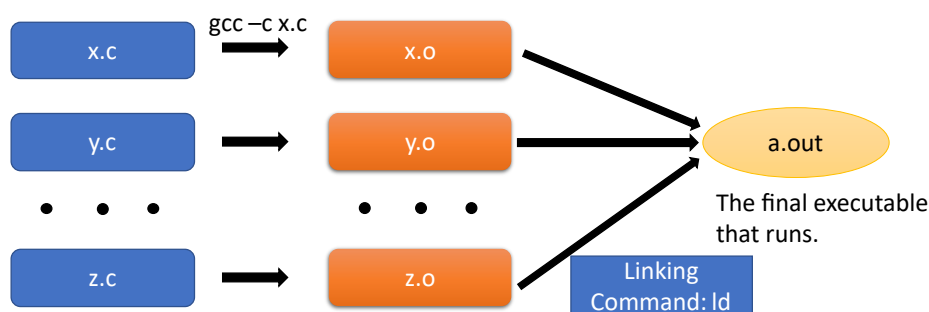
Figure B.1: The process of compiling and linking

Overview of Header Files

It turns out that there is another serious problem. Whenever, a function is invoked in a C file, we need to know its *signature* – the number of arguments, their respective data types and the data type of the return value. If the C function's signature is not available to the compiler, then it will not be able to generate code or even check whether the program has been written correctly or not. This is bound to be the case when functions are defined in other C files. Languages such as C furthermore also automatically change the type (typecasting) of input arguments based on the signature of the function. For instance, if a character (char) type variable is sent to a function that expects an integer, then automatic type conversion takes place. There are many more such cases where a similar type conversion scheme is used. But to do that, we need to insert dedicated code in the compiled program and thus knowing the signature of the function is essential. Once the signature is provided, the original function could be defined in some other C file, which per se is not an issue – we can compile the individual C source code file seamlessly. To summarize, the signature of a function needs to be available at the time of compilation.

> **Point B.1.2**
>
> The signature of a function can be used to implement many tasks like automatic type conversion, correctly arranging the arguments and properly typecasting the return value. Without the signature, the compilation process will fail. Hence, the signatures of externally defined functions should be available to C files. This will allow them to use a function without defining it.

Let us further delve into the problem of specifying function signatures, which will ensure that we can at least compile a single C source code file correctly and create the corresponding object file. Subsequently, the linker can combine all the object files and create the program's binary or executable.

B.1.3 The Concept of the Header File

The direction of our discussion centers around specifying the signatures of functions even though the functions may themselves be defined somewhere else. Let us thus introduce two terms – declaration and definition. The *declaration* refers to specifying the signature of a function. For instance, a declaration may look like this: `int foo (int x, char y, double z);`. On closer examination, we note that there is no need to specify the names of the parameters because this information is not of any use. All that the compiler needs to know are the types of the parameters and the type of the return value. Hence, an alternative signature can be `int foo(int, char, double);`. This is all that is required.

Novice programmers normally specify the signature of all functions at the beginning of a C file or right before the function is actually used. This will do the job, even though it is not an ideal solution. Often the keyword *extern* is used in C and C++ programs to indicate that the function is actually defined somewhere else.

The *definition* refers to the function's actual code – C statements within the function. Consider a project with a single C file, where a function is invoked after

it is defined. In this case, there is no need to actually declare the signature of the function – the definition serves the purpose of also declaring the signature of the function. However, in the reverse case, a declaration is necessary. For example, let us say that the function is invoked in Line 19 and its code (definition) starts at Line 300. There is a need to declare the signature of the function before Line 19. This is because when the relevant compilation pass processes Line 19, it will already be armed with the signature of the function, and it can generate the corresponding code for invoking the function correctly.

We need to do something similar for functions defined in other files in a large multifile project. Of course, dealing with so many signatures and specifying them in every source code file is a very cumbersome process. In fact, we also have to specify the signature of global variable definitions (their types) and even enumerations, structures and classes. Hence, it makes a lot of sense to have a dedicated file to just store these signatures. There can be a pre-compilation phase where the contents of this file are copy-pasted into source code files (C or C++ files).

A header file or a .h file precisely does this. It contains a large number of signatures of variables, functions, structs, enumerations and classes. All that a C file needs to do is simply include the header file. Here the term `include` means that a pre-compilation pass needs to copy the contents of the header file into the C file that is including it. This is a very easy and convenient mechanism for providing a bunch of signatures to a C file. For instance, there could be a set of C files that provide cryptographic services. All of them could share a common header file via which they export the signatures of the functions that they define to other modules in a large software project. Other C files need to include this header file and call the relevant functions defined in it to obtain cryptographic services. The header file thus facilitates a logical grouping of variable, function and structure/class declarations. It is much easier for programmers to include a single header file that provides a cohesive set of declarations as opposed to manually adding declarations at the beginning of every C file.

Header files have other interesting uses as well. Sometimes, it is easier to simply go through a header file to figure out the set of functions that a set of C functions provide to the rest of the world. It is a great place for code browsing.

Barring a few exceptions, header files never contain function definitions or any other form of source code. Their role is not to have regular C statements. This is the role of source code files such as .c and .cpp files. Header files are reserved only for signatures that aid in the process of compilation. For the curious reader, it is important to mention that the only exception to this rule is C++ templates. A template is basically a class definition that takes another class or structure as an argument and generates code based on the type of the class that is passed to it at compile time.

Now, let us look at a set of examples to understand how header files are meant to be used.

Example

Listing B.1: factorial.h

```
#ifndef FACTORIAL_H
#define FACTORIAL_H
```

```
extern int factorial (int);
#endif
```

Listing B.1 shows the code for the header file `factorial.h`. First, we check if a preprocessor variable `FACTORIAL_H` is already defined. If it is already defined, it means that the header file has already been included. This can happen for a variety of reasons. It is possible that some other header file has included `factorial.h`, and that header file has been included in a C file. Given that the contents of `factorial.h` are already present in the C file, there is no need to include it again explicitly. This is ensured using preprocessor variables. In this case, if `FACTORIAL_H` has not been defined, then we define the function's signature: `int factorial(int);`. This basically says that it takes a single integer variable as input and the return value is an integer.

Listing B.2: factorial.c

```
#include "factorial.h"

int factorial (int val){
    int i, prod = 1;
    for (i=1; i<= val; i++) prod *= i;
    return prod;
}
```

Listing B.2 shows the code of the `factorial.c` file. Note the way in which we are including the header file. It is being included by specifying its name in between double quotes. This normally means that the header file should be there in the same directory as the C file (factorial.c). We can also use the traditional way of including a header file between the '<' and '>' characters. In this case, the directory containing the header file should be there in the `include path`. The "include path" is a set of directories in which the C compiler searches for header files. The directories are searched in ascending order of preference based on their order in the include path. There is always an option of adding an additional directory to the include path by using the '-I' compilation flag in `gcc`. Any directory that succeeds the '-I' flag is made a part of the include path and the compiler searches that directory as well for the presence of the header file. Now, when the compiler compiles `factorial.c`, it can create `factorial.o` (the corresponding object file). This object file contains the compiled version of the factorial function.

Let us now try to write the file that will use the factorial function. Let us name it `prog.c`. Its code is shown in listing B.3.

Listing B.3: prog.c

```
#include <stdio.h>
#include "factorial.h"

int main(){
    printf("%d\n",factorial(3));
}
```

All that the programmer needs to do is include the `factorial.h` header file and simply call the `factorial` function. The compiler knows how to generate the code for `prog.c` and create the corresponding object file `prog.o`. Given

that we have two object files now – `prog.o` and `factorial.o` – we need to *link* them together and create a single binary that can be executed. This is the job of the linker that we shall see next. Before we look at the linker in detail, an important point that needs to be understood here is that we are separating the signature from the implementation. The signature was specified in `factorial.h` that allowed `prog.c` to be compiled without knowing how exactly the `factorial` function is implemented. The signature had enough information for the compiler to compile `prog.c`.

In this mechanism, the programmer can happily change the implementation as long as the signature is the same. The rest of the world will not be affected, and they can continue to use the same function as if nothing has changed. This allows multiple teams of programmers to work independently as long as they agree on the signatures of functions that their respective modules export.

B.2 Linker

The role of the linker is to combine all the object files and create a single executable. Any project in C/C++ or other languages typically comprises multiple source files (.c and .cpp). Moreover, a source file may use functions defined in the standard library. The standard library is a set of object files that defines functions that many programs typically use such as `printf` and `scanf`. The final executable needs to <u>link</u> these library files (collections of object files) as well.

> **Definition B.2.1** Standard C Library
>
> The standard C library is a collection of compiled functions that enable a user program to access system services such as reading and writing to files or sending messages over the network.

There are two ways of linking: static and dynamic. Static linking is a simple approach where we just combine all the .o files and create a single executable. This is an inefficient method as we shall quickly see. This is why dynamic linking is used where all the .o files are not necessarily combined into a single executable at the time of linking.

B.2.1 Static Linking

A simple process of compilation is shown in Figure B.2. In this case we just compile both the files: `prog.c` and `factorial.c`. They can be specified as arguments to the `gcc` command, or we can separately create .o files and then compile them using the `gcc` command. In this case, the `gcc` command invokes the linker as well.

The precise role of the linker is shown in Figure B.3. Each object file contains two tables: the symbol table and the relocation table. The symbol table contains a list of all the symbols – variables and functions – defined in the .o file. Each entry contains the name of the symbol, sometimes its type and scope, and its address. The relocation table contains a list of symbols whose address has not been determined as yet. Let us now explain the linking process that uses these tables extensively.
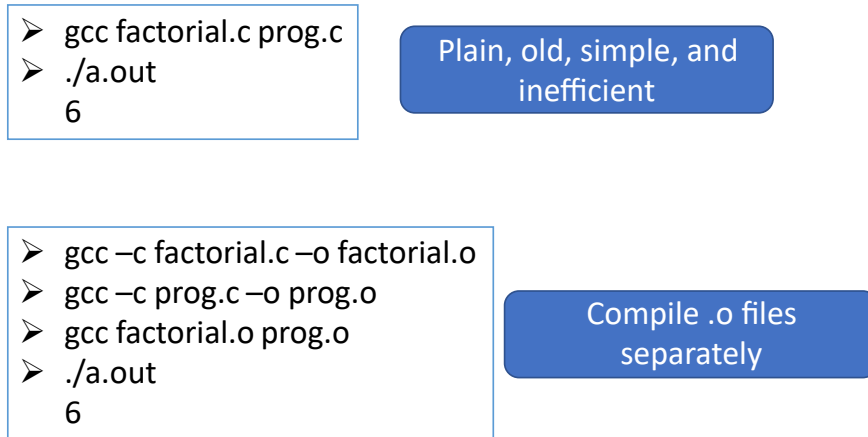
Figure B.2: Code for static linking
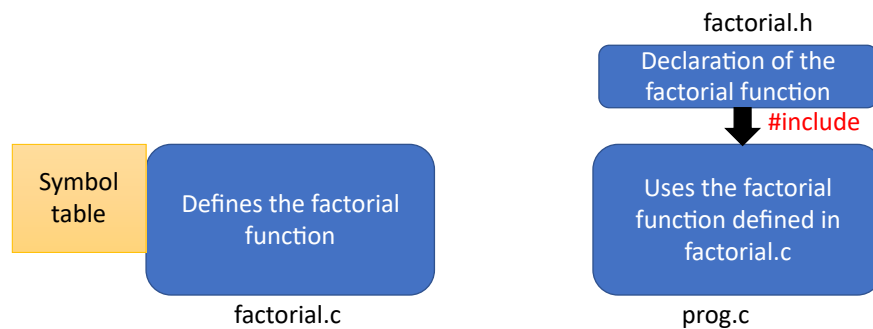
Figure B.3: Compiling the code in the factorial program and linking the components

Each object file contains some text (program code), read-only constants and global variables that may or may not be initialized. Along with that it references variables and functions that are defined in other object files. All the symbols that an object file exports to the world are defined in the symbol table and all the symbols that an object file needs from other object files are listed in the relocation table. The linker thus operates in two passes.

Pass 1: It scans through all the object files and concatenates all the text sections (instructions), global variables, function definitions and constant definition sections. It also makes a list of all the symbols that have been defined in the object files. This allows the linker to compute the final sizes of all the sections: text, data (initialized global/static variables), bss or (block starting symbol, uninitialized global/static variables) and constants. All the program code and variable definitions are concatenated and the final addresses of all the variables and functions are computed. The concatenated code is however incomplete. The addresses of all the relocated variables and functions (defined in other object files) are set to

zero (undefined).

Pass 2: In this stage, the addresses of all the relocated variables and functions are set to their real values. We know the address of each variable at the end of Pass 1. In the second pass, the linker replaces the zero-valued addresses of relocated variables and functions with the actual addresses computed in the first pass.

Issues with Static Linking

Let us now look at some common issues with static linking. In this case, we want to link all the object files as well as the standard C library together and build one large executable. This is shown in Figure B.4. We quickly observe that to statically link everything, all that we need to do is add the '-static' flag to the compilation options. We can check this using the `ldd` command. The output will show that the executable is statically linked, and no dynamic libraries are referenced. For a very simple program that simply prints integers, the size of the executable is quite large. It is close to 1 MB (892 KB to be precise on your author's system). There are several reasons for this. The first is that the code that we write is not ready by itself to be executed by the operating system. The compiler typically adds many more functions that setup all the memory regions, load the constants into memory and create the environment for program execution.

The first function to be called is `_start`. It starts the process of setting up the memory space of the process. It invokes a sequence of functions: one of them is `__libc_start_main`, which ultimately calls the `main` function. The code of all these functions needs to be present in the executable. The `main` function is thus not the first function to be invoked. Even when the main function returns, the process does not immediately terminate. Again a sequence of functions are invoked that release all the resources that the process owned such as open files and network connections.

Surprisingly, all this overhead is not much. The dominant source of overheads here is the code of all the C libraries that is added to the executable. This means that if we invoke the `printf` function, then the code of `printf` as well as the set of all the library functions that `printf` invokes (and in turn they invoke) are added to the executable. These overheads can be quite prohibitive. Assume that a program's source code contains one hundred unique library calls, but in any practical execution only 25 unique library calls are made. The size overhead is 100/25 (4×). Sadly, at compile time, we don't know which library calls will be made and which ones will not be made. Hence, we conservatively assume that every single library call that is mentioned in any object file will actually be made, and it is not dead code. Consequently, the code for all these library functions (including their backward slice) needs to be included in the executable. Here, the *backward slice* of a library function such as `printf` comprises the set $\mathcal{S}$ of library functions called by `printf`, as well as all the library functions invoked by functions in $\mathcal{S}$, so on and so forth. Formally, this set is known as the reflexive-transitive closure of the `printf` function. Because of this, we need to include a lot of code in executables. Therefore, they become very large. This can be visualized in Figure B.4.

Along with the large size of executables, which in itself is problematic, we

lose a chance to reuse code pages that are required by multiple processes. For instance, almost all processes share a few library functions defined in the standard C library. As a result, we would not like to replicate the code pages of library functions – this would lead to a significant wastage of memory space. Hence, we would like to share them across processes saving a lot of runtime memory.

To summarize, if we use such statically linked binaries where the entire code is packaged within a single executable, such code reuse options are not available to us. Hence, we need a better solution. This solution is known as dynamic linking.
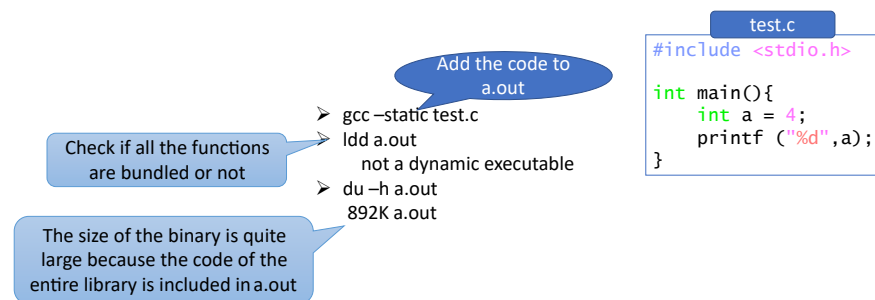
Figure B.4: Size of a statically linked executable

B.2.2   Dynamic Linking

Dynamic linking solves many of the problems with static linking. The basic idea here is that we do not add the code of all library functions or even functions defined in object files (part of the program's code base) unless there is a very high chance that the code will actually be used and that too very frequently. Furthermore, we would also not like to add code to an executable if there is a high chance that it will be reused across many processes. If we follow these simple rules, the size of the binary will remain reasonably small. However, the program execution gets slightly complicated because now there will be many functions whose code is not a part of the executable. As a result, invoking those functions will involve a certain amount of complexity. Some of this is captured in Figure B.5.

In this case, where `printf` is dynamically linked, the address of the `printf` symbol is not resolved at link time. Instead, the address of `printf` is set to a dummy function known as a *stub function*. The first time that the stub function is called, it locates the path of the library that contains the `printf` function, then it copies the code of the `printf` function to a memory address that is within the memory map of the process. Finally, it stores the address of the first byte of the `printf` function in a dedicated table known as the *jump table*. The next time the stub function is called, it directly accesses the address of the `printf` function in the jump table. This basically means that the first access to the `printf` function is slow. Henceforth, it is very fast.

The advantages of this scheme are obvious. We only load library functions on-demand. This minimizes the size of the executable. Furthermore, we can
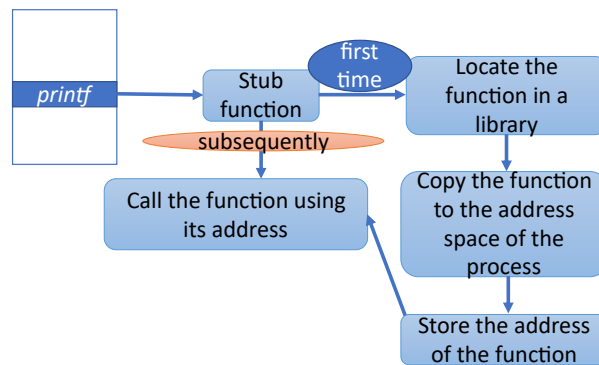
Figure B.5: Dynamically linking a program

have one copy of the shared library code in physical memory and simply map regions of the virtual address space of each process to the physical addresses corresponding to the library code. This also minimizes the memory footprint and allows as much of runtime code reuse as possible. Of course, there is a very minor performance penalty. Whenever a library function is accessed for the first time, there is a necessity to first search for the library first and then find the address of the function within it. Searching for a library, proceeds in the same manner as searching for header files.

During the process of compilation, a small note is made about which function is available in which library. Now if the executable is transferred to another machine and run there or even run on the same machine, it is necessary to locate the library at runtime. The stub function calls a function named `dlopen`. When invoked for the first time for a given library function, its job is to locate the library. Akin to the way that we search for a header file, there is a search order. We first search for the library in the current directory. If it is not found, we check the directories in the *LD_LIBRARY_PATH* environment variable. Then we search known locations in the system such as `/lib` and `/usr/lib`. The search order is very important because often there are multiple copies of a library, and we want the program to fetch the correct copy.

Each library defines a symbol table that lists the symbols that it exports to the rest of the world. This is how we can find the addresses of the functions that are present within the library and copy them to the memory space of the process that dynamically links the library. The code can also be copied to a shared location and then mapped to the virtual address space of any process that wishes to use the code. This is a very efficient method and as of today, this is the de facto standard. Almost all software programs use the shared library based dynamic linking mechanism to reduce their code size and ensure that they remain portable across systems.

Many times, when we are not sure if the target system has a shared library or not, the software package can either bundle the shared library along with the executable or the target system can install the shared library first. This is very common in Linux-based systems, where shared libraries are bundled into packages. Whenever, a software is installed (also distributed as a package), it checks for dependencies. If a package is dependent on other packages, then

it means that those packages provide some shared libraries that are required. Hence, it is necessary to install them first. Moreover, these packages could have dependencies with other packages that also need to be installed. We thus need to compute the *backward slice* of a package and install the missing packages. This is typically done by the package manager in Ubuntu or RedHat Linux.

It is important to note that the notion of shared libraries and dynamic linking is there in all operating systems, not just Linux. For example, it is there in Windows where it is known as a DLL (dynamically linked library). Conceptually, a shared library on Linux (.so file) and a DLL in Windows (.dll file) are the same.

```
➢ gcc –c factorial.c –o factorial.o
➢ ar –crs factorial.a factorial.o
➢ gcc prog.o factorial.a
➢ ./a.out
    6
```

The *ar* command creates a library out of several .o files

factorial.a is a library that is statically linked in this case

```
➢ gcc –c –fpic –o factorial.o factorial.c
➢ gcc –shared –o libfactorial.so factorial.o
➢ gcc –L. prog.c -lfactorial
➢ export LD_LIBRARY_PATH=`pwd`
➢ ./a.out
    6
```

Generate position independent code
Create the shared library
Create the executable. Reference the shared library.
Tell the system that the factorial library is in the current directory

Figure B.6: Code for dynamic linking

Figure B.6 shows the method to generate a shared object or shared library in Linux. In this case, we want to generate a shared library that contains the code for the factorial function. Hence, we first compile the `factorial.c` file to generate the object file (`factorial.o`) using the '-c' `gcc` option. Then we create a library out of the object file using the archive or `ar` command. The extension of the archive is '.a'. This is a static library that can only be statically linked like a regular .o file.

The next part shows us how to generate a dynamic library. First, we need to compile the `factorial.c` file in a way that is position independent – the starting address of the code does not matter. This allows us to place the code at any location in the virtual address space of a process. All the addresses are relative to a base address. In the next line, we generate a shared object from the `factorial.o` object file using the '-shared' flag. This generates `libfactorial.so`. Next, we compile and link `prog.c` with the dynamic library that we just created (`libfactorial.so`). This part is tricky. We need to do two separate things.

Consider the command `gcc -L. prog.c -lfactorial`. We use the '-L' flag to indicate that the library will be found in the current directory. Then, we specify the name of the C file, and finally we specify the library using the '-l' flag. Note that there is no space in this case between '-l' and `factorial`. The compiler searches for `libfactorial.so` in the current directory because of the -L and -l flags.

In this case, running the executable `a.out` is not very straightforward. We need to specify the location at which the factorial library will be found given that it is not placed in a standard location that the runtime (library loader)

usually checks such as a `/lib` or `/usr/lib`. We thus add the current directory (output of the `pwd` command) to the LD_LIBRARY_PATH environment variable. After that we can seamlessly execute the dynamically linked executable – it will know where to find the shared library (`libfactorial.so`).

Readers are welcome to check the size of dynamically linked executables. Recall the roughly 1 MB sized executable that we produced post static linking (see Figure B.4); its size reduces to roughly 12 KB with dynamic linking !!!

Let us finish this round of discussion with describing the final structure of the executable. After static or dynamic linking, Linux produces a shared object file or executable in the ELF format.

B.2.3 The ELF Format

The ELF format (Executable and Linkable Format) is arguably the most popular format for executables and shared libraries. An executable or a shared library in the ELF format starts with a header that describes the structure of the file and details about the ISA and machine compatibility. It is divided into several sections: contiguous regions in the virtual address space that store the same type of information (code, constants, etc.). Sections are grouped into segments. An ELF executable or binary has a program header table (list of segments) and a section header table (list of sections). The starting address of each section or segment is specified in these tables.

The typical sections in an ELF file are the text section (instructions in the binary), data section (initialized data), bss section (uninitialized data), symbol table (list of variables and functions defined in the file) and the relocation table (variables and functions that are defined elsewhere). There is some information regarding dynamic linking in the corresponding section (dynamic).

B.3 Loader

The loader is the component of the operating system whose job is to execute a program. When we execute a program in a terminal window, a new process is spawned that runs the code of the loader. The loader reads the executable file from the file system and lays it out in main memory. It needs to parse the ELF executable to realize this.

It creates space for all the sections, loads the constants into memory and allocates regions for the stack, heap and data/bss sections (static and global variables). Additionally, it also copies all the instructions into memory. If they are already present in the memory system, then instead of creating a new copy, we can simply map the instructions to the virtual memory of the new process. If there is a need for dynamic linking, then all the information regarding dynamically linked symbols is stored in the relocation table and the dynamic section in the process's memory image. The loader also initializes the jump tables.

Next, it initializes the execution environment such as setting the state of all the environment variables, copying the command line arguments to variables accessible to the process and setting up exception handlers. Sometimes for security reasons, we wish to randomize the starting addresses of the stack and heap such that it is hard for an attacker to guess runtime addresses. This can

be done by the loader. It can generate random values within a pre-specified range and initialize base addresses in the program such as the starting value of the stack pointer and the heap memory region.

The very last step is to issue a system call to erase the memory state of the loader and start the process from the first address in its text section. The process is now alive, and the program is considered to be *loaded*.

Appendix C

Data Structures

In this section, we provide an overview of the commonly used data structures in the Linux operating system. Note that this is not meant to be a rigorously theoretical section. Given that there are excellent texts on algorithms and data structures [Cormen et al., 2009], there is no need to rigorously explain all the concepts in detail in this book. The main aim of this appendix is to briefly describe the data structures, provide some examples, list their main properties and highlight where these data structures are used. At the end of this short appendix, the reader should clearly be able to understand when and where a data structure should be used and what it is good for. Furthermore, the reader should be able to appreciate the limitations of every data structure and why a need may arise to mix-and-match a bunch of data structures to solve a real-world problem.

C.1   Linked Lists in Linux

Defining generic link lists in the kernel code is a problem of fundamental importance. Note that the way that we normally define linked lists, which is by declaring a structure of the form – `struct Node` – and then having a member called `struct Node *next` is not a great idea. It will not lead to a generic solution where we can create a linked list out of diverse structures. This is because if we want to write generic routines to operate on linked lists, then they will have to take a generic `void *` pointer as an argument for the encapsulating object (structure). A need will arise to find a pointer to the `next` member. The default solution is to typecast the `void *` pointer to a pointer to the type of object that it points to. However, this solution will not lend itself to a generic implementation because the code to traverse a linked list should be independent of the type of the linked list node. In a programming language that supports templates where the type of the node is sent as an argument in the code, implementing generic routines is very easy. The C++ standard library uses such methods. This facility is sadly not available in languages such as C that do not have such sophisticated features.

In any large code base like the Linux kernel, we have linked lists for all kinds of structures and thus a method is required where it is very easy to operate on

such linked lists as well as easily create a linked list out of any kind of structure. This is a very important software engineering problem that the early developers of the kernel faced. Given that the kernel is written in C, a novel solution had to be created.

Linux's solution is quite ingenious. It heavily relies on C macros, which are unique strength of C. We would advise the reader to go through this topic before proceeding forward. Macros are very useful yet very hard to understand.

C.1.1 struct list_head

Listing C.1 shows the definition of `struct list_head` – the data structure representing a doubly linked list. This structure has two members, which are pointers to lists of the same type `struct list_head`. They are named `next` and `prev`, respectively. This is all that there is to the definition of a linked list. A `struct list_head` structure represents a linked list node – it has pointers to the next and previous entries. This is surprisingly enough to operate on the linked list. For example, we can traverse the list, add new entries as well as remove entries.

The crucial question that we need to answer here is, "Where is the encapsulating object (structure) that needs to be linked together?" In general, we define a linked list in the context of an object (such as `struct Node`). We interpret the linked list to be a list of such objects. Here, there is no such object. Instead, there is just a generic linked list node with pointers to its next and previous nodes. It is not storing any other information of interest and thus this solution does not appear to be very useful. It is true that it satisfies our demand for generality; however, it does not align with our intuitive notion of a linked list as we have studied in a data structures course.

Listing C.1: The definition of a linked list
source : include/linux/types.h#L178

```
struct list_head {
    struct list_head *next, *prev;
}
```

This is where we will use the magic of macros. We will use two macros to solve this problem as shown in Listing C.2.

Listing C.2: The `list_entry` and `container_of` macros
source : include/linux/list.h#L519 and
source : include/linux/container_of.h#L18 (resp.)

```
    #define list_entry (ptr, type, member)  container_of (
        ptr, type, member)

    #define container_of(ptr, type, member) ({
        void *__mptr = (void *)(ptr);
        ((type *)(__mptr - offsetof(type, member))); })
```

Focus on the `container_of` macro. It takes three inputs: a pointer, a type and a member name. The first statement simply typecasts the pointer to `void*`. This is needed because we want to create a generic implementation, which is not dependent on any particular type of object. The `offsetof macro` provides the offset of the starting address of the member from the beginning of the structure.

Listing C.3: Examples of structures

```
struct abc {
    int x;
    struct list_head list;
}
struct def {
    int x;
    float y;
    struct list_head list;
}
```

Consider the structures shown in Listing C.3. In the case of `struct abc`, the value of `offsetof(abc, list)` is 4. This is because we are assuming the size of an integer is four bytes. The integer `x` is stored in the first four addresses of `struct abc`. Hence, the offset of the `list` member is 4 here. On the same lines, we can argue that the offset of the member `list` in `struct def` is 8. This is because the size of an integer and that of a float are 4 bytes each. Hence, (`_mptr - offsetof(type, member)`) provides the starting address of the structure that is the linked list node. To summarize, the `container_of` macro returns the starting address of the linked list node or in other words the encapsulating object given the offset of the `list` member in the object.

It is important to note that this is a compile-time operation. Specifically, it is the role of the preprocessor to execute macros. The preprocessor is aware of the code as well as the layouts of all the structures that are used. Hence, for it to find the offset of a given member from the starting address of a structure is very easy. After that computing the starting address of the linked list node (encapsulating object) is easy. This is a simple piece of code that the macro will insert into the program. It involves a simple subtraction operation.

A macro is quite different from a regular function. Its job is to generate custom code that is subsequently compiled. In this case, an example piece of code that will be generated will look like this: (`Node *)(_mptr - 8)`. Here, we are assuming that the structure is `struct Node` and the offset of the `list` member within it is 8. At runtime, it is quite easy to compute this given a pointer (`ptr`) to a `struct list_head`.

Listing C.4: Example of code that uses the `list_entry` macro

```
struct abc* current = ... ;
struct abc* next = list_entry (current ->list.next , struct
    abc , list);
```

Listing C.4 shows a code snippet that uses the `list_entry` macro where `struct abc` is the linked list node. The `list_entry` macro is simply a synonym of `container_of` – their signatures are identical. The current node that we are considering is `current`. To find the next node (the next one after `current`), which is again of type `struct abc`, all that we need to do is invoke the `list_entry` macro. In this case, the pointer (`ptr`) is `current->list.next`. This is a pointer to the `struct list_head` object in the next node. From this pointer, we need to find the starting address of the encapsulating `abc` structure. The type is `struct abc` and the member is `list`. The `list_entry` macro internally calls `offsetof`, which returns an integer. This integer is subtracted from

the starting address of the `struct list head` member in the next node. The final result is a pointer to the encapsulating object.

Such a mechanism is a very fast and generic mechanism to traverse linked lists in Linux. It is independent of the type of the encapsulating object. These primitives can also be used to add and remove nodes from the linked list. We can extend this discussion to create a linked list that has different kinds of encapsulating objects. Theoretically, this is possible as long as we know the type of the encapsulating object for each `struct list head` on the list.

C.1.2 Singly-Linked Lists

Listing C.5: The `hlist` based singly-linked list
source : include/linux/types.h#L182

```
struct hlist_head {
    struct hlist_node *first;
};

struct hlist_node {
    struct hlist_node *next, **pprev;
};
```

Let us now describe singly-linked lists that are frequently used in kernel code. Here the explicit aim is a one-way traversal of the linked list. An example is a hash table where we resolve collisions by chaining entries that hash to the same entry. Linux uses the `struct hlist head` structure (shown in Listing C.5). It points to a node that is represented using `struct hlist node`.

This data structure has a `next` pointer to another `hlist node`. Sadly, this information is not enough if we wish to delete the `hlist node` from the linked list. We need a pointer to the previous entry as well. This is where a small optimization is possible, and a few instructions can be saved. We actually store a pointer to the `next` member of the previous node in the linked list. This information is stored in the field `pprev`. Its type is `struct hlist node **`. The advantage of this is that we can directly set it to a different value while deleting the current node. We cannot do anything else easily, which is the explicit intention here. The conventional solution in this case is to store a pointer to the previous `hlist node`. Any delete method needs to first fetch this pointer, compute the address of its `next` member, and then reassign the pointer to a different value. The advantage of the `pprev` pointer is that we save on the instruction that computes the address of the `next` pointer of the previous node.

Such data structures that are primarily designed to be singly-linked lists are often very performance efficient. Their encapsulating objects are accessed in exactly the same way as the doubly-linked list `struct list head`.

C.2 Red-Black Tree

A red-black (RB) tree is a very efficient data structure for searching data – it is a special kind of BST (binary search tree). As the name suggests, there are two kinds of nodes: red and black. It is a balanced binary search tree, where it is

possible to insert, delete and search items in logarithmic time. We can ensure all of these desirable properties of the tree by following these simple rules:

1. A node is either red or black.

2. The leaf nodes are *special*. They don't contain any data. However, they are always presumed to be black. They are also referred to as *sentinel nodes*.

3. A red node never has a red child. Basically, red nodes are never adjacent.

4. Any path from the root to any leaf node has the same *black depth*. The *black depth* of a leaf node is defined as the number of black nodes that are crossed while traversing the tree from the root to the leaf node. In this case, we are including both the root and the leaf node.

5. If a node has exactly one non-leaf child, then the child's color must be red.

Traversing a red-black tree is quite simple. We follow the same algorithm as traversing a regular binary search tree. The claim is that the tree is balanced – the height of the tree is $O(log(n))$. Specifically, the property that this tree guarantees is as follows.

> **Point C.2.1**
>
> The maximum depth of any leaf is at most twice the minimum depth.

This is quite easy to prove. As we have mentioned, the black depth of all the leaves is the same. Furthermore, we have also mentioned that a red node can never have a red child. Assume that in any path from the root to a leaf, there are r red nodes and b black nodes. We know that b is a constant for all paths from the root. Furthermore, every red node will have a black child (note that all leaves or sentinel nodes are black). Hence, $r \leq b$. The total depth of any leaf is $r + b \leq 2b$. This basically means that the maximum depth is at most twice the minimum depth b.

This vital property ensures that all search operations always complete in $O(log(n))$ time. Note that a search operation in an RB tree operates in exactly the same manner as a regular binary search tree. Insert and delete operations also complete in $O(log(n))$ time. They are however not very simple because we need to ensure that the black depth of all the leaves always stays the same, and a red parent never has a red child.

This requires a sequence of recolorings and rotations. However, we can prove that at the end, all the properties hold and the overall height of the tree is always $O(log(n))$.

C.3 B-Tree

A B-tree is a generalization of a binary search tree. It is a k-ary tree and is self-balancing. In this case, a node can have more than two children; quite unlike a red-black tree. This is also a balanced tree and all of its operations are realizable in logarithmic time. The methods of traversing the tree are very

similar to traversing a classical binary search tree. It is typically used in systems that store a lot of data and quickly accessing a given datum or a contiguous subset of the data is essential. Hence, databases and file systems tend to use B-trees quite extensively.

Let us start with the definition of a B-tree of order m. It stores a set of keys, which can optionally point to values. The external interface is similar to a hash table.

1. Every node has at most m children.

2. The root needs to contain at least one key.

3. It is important that the tree does not remain sparse. Hence, every internal node needs to have at least $\lceil m/2 \rceil$ children (alternatively $\lceil m/2 \rceil - 1$ keys).

4. If a node has m children, it needs to store $m - 1$ keys. These $m - 1$ keys partition the space of keys into m non-overlapping regions. Each child stores keys for the region that belongs to it. It is assigned a key space.

5. All the leaves are the same level.

C.3.1   The Search Operation
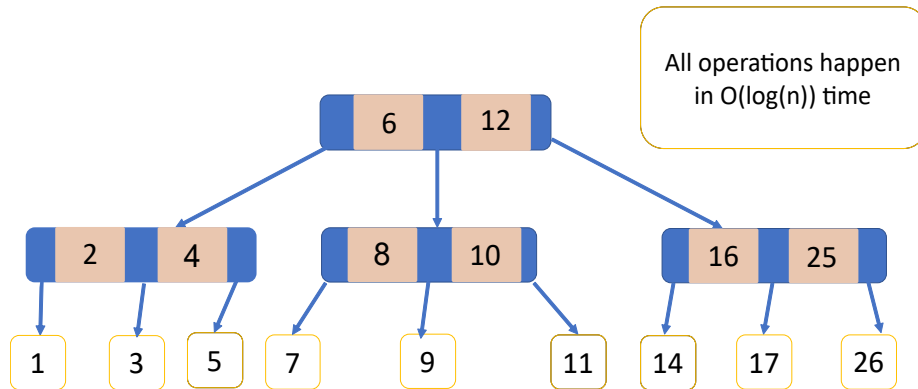
Figure C.1: Example of a B-tree

Figure C.1 shows an example of a B-tree of order 3 ($m = 3$). In a B-tree, we also store the values associated with the keys. These values could be stored in the node itself or there could be pointers within a node to point to the values corresponding to its keys. There are many ways of implementing this. We shall not focus on this aspect because the storage of values is not central to the operation of a B-tree.

In the example, consider the root node. It stores two keys: 6 and 12. All the keys less than 6 are stored in the leftmost child. The leftmost child is an internal node, which stores two keys: 2 and 4. Note that both of them are less than 6. They point to leaf nodes that store a single key each. Note that as per our definition (order=3), this is allowed. Key 1 is less than 2; hence, it is the leftmost child. Key 3 is stored in the middle child (between 2 and 4). Finally,

Key 5 is stored in the rightmost child. The same logic applies for the second child of the root node. It needs to store keys that are strictly greater than 6 and less than 12. We again see a similar structure with the internal node that stores two keys – 8 and 10. It points to three leaf nodes. Finally, the rightmost child of the root only stores keys that are greater than 12.

It is easy to observe that traversing a B-tree is similar to traversing a regular BST (binary search tree). It has $O(log_m(n))$ levels. At each level, the time to find the pointer to the right subtree takes $O(log(m))$ time. We are assuming that we perform a binary search over all the keys. The total time complexity is thus $O(log_m(n)log(m))$, which is $O(log(n))$.

C.3.2 The Insert and Delete Operations

In the case of an insert operation, we can traverse the tree from the root till a leaf. Given that internal nodes can store keys, we can first try to store the key in an internal node if it has adequate space. Otherwise, the next option is to store it in a leaf assuming that there is space in the leaves. If this process is not successful, then we may have to split an internal node into two nodes and add the key to one of them. There would be a need to remove one of the keys from them and add it to the parent node. Adding a key to the parent node is important because we need to split the corresponding key subspace into two parts.

We need to understand that all these operations do not change the depth of any leaf. In the worst case, when this cannot be done, a need will arise to split the root, create two internal nodes and initialize a new parent. This will also ensure that the depth of all the leaves is the same. It is just that the height of the tree will increase by one.

Deletion is the reverse process. In this case, we can remove the key as long as the node still has $\lceil m/2 \rceil - 1$ keys left in it. However, if this is not the case, then a need will arise to merge two adjacent sibling nodes and move the key separating the internal nodes from the parent to the merged node. This is pretty much the reverse of what we did while adding a new key. Here again, a situation will arise when this cannot be done, and we will be forced to reduce the height of tree.

The time complexity of both of these operations is $O(log(n))$.

C.3.3 B+ Tree

The B+ tree is a variant of the classical B-tree. In the case of a B-tree, internal nodes can store both keys and values, however in the case of a B+ tree, internal nodes can only store keys. All the values (or pointers to them) are stored in the leaf nodes. Furthermore, all the leaf nodes are connected to each other using a linked list, which allows for very efficient range queries. It is also possible to do a sequential search in the linked list and locate data with proximate keys quickly.

C.3.4 Advantage of B-Trees and B+ Trees

Let us now look at the advantages of these structures. Given that the asymptotic time complexity is the same for binary search trees, B-trees and B+ trees,

i.e., $(O(log(n)))$, the advantages of these structures arise due to efficient cache behavior.

A balanced binary search tree (BST) has roughly $log_2(n)$ levels, whereas a B-tree and its variants have $log_m(n)$ levels. They thus have fewer levels mainly because more information is stored in each internal node. This is where the design can be made cache efficient. An internal node can be designed in such a way that its contents fit within a cache block or maybe a few cache blocks. The node's contents fully occupy a cache block and no other information is stored in each cache block. The advantages of these schemes are thus plenty. We end up fetching fewer cache blocks to traverse the tree as compared to a BST. This is because a cache block fetch is more productive. There is much more information in a block in a B-tree. Fetching fewer cache blocks is a good idea. Statistically, there will be fewer cache misses and the chances of having long memory-related stalls will be much lower.

It is important to understand that a 64 or 128-byte cache block is the atomic unit of transfer in the memory system. There is no point in fetching 64 bytes yet using only 25% of it as is the case in a BST that simply stores two pointers in each node: one to the left child and one to the right child.

There are other advantages as well. We ideally do not want the data of two different tree nodes to be stored in the same cache block. In this case, if different threads are accessing different nodes in the tree and making write accesses, there is a chance that they may actually be accessing the same cache block. This will happen in the case of a BST and will not happen with a B-tree. Due to such conflicting accesses, there will be a lot of misses due to cache coherence in a BST. The cache block will keep bouncing between cores. Such misses are known as *false sharing misses*. Note that the same data is not being shared across threads. The data is different, yet they are resident in the same cache block. This problem does not afflict a B-tree and its variants.

Along with reduced false sharing, it is easy to handle true sharing misses as well. In this case, two threads might be trying to modify the same tree node. It is possible to lock a node quite easily. A small part of the corresponding cache block can be reserved to store a multi-bit lock variable. This makes acquiring a "node lock" very easy.

For a combination of all these factors, B-trees and B+ trees are preferred as compared to different flavors of balanced binary search trees.

C.4 Maple Tree

The maple tree is a data structure that is commonly used in modern versions of Linux kernels [Rybczynska, 2021, Howlett, 2021]. It is a variant of the classical B+ tree with additional restrictions. The maple tree used in the Linux kernel has hardwired branching factors (max. number of children per node). A non-leaf node can store a maximum of 10 children (9 keys). Leaf nodes can store up to 15 entries. They don't have any children.

The nodes are aligned to cache line boundaries. This eliminates misses due to false-sharing [Sarangi, 2023]. Furthermore, it is possible to service concurrent accesses – multiple users can seamlessly operate on different parts of the maple tree. Each key can either be a single value or can be a range, as is the case for VM regions (key = *start* and *end* addresses).

C.5 Radix Tree
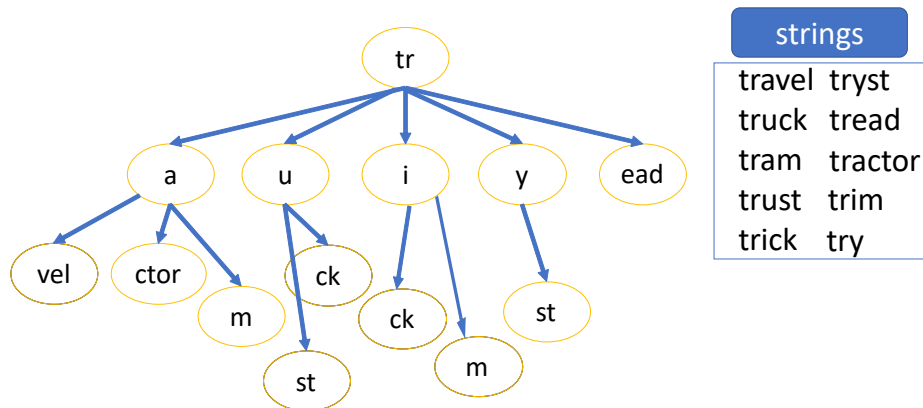
Figure C.2: Example of a radix tree

A radix tree stores a set of keys very efficiently. Each key is represented as a string (see Figure C.2). The task is to store all the keys in a single data structure, and it is possible to query the data structure and find if it contains a given string (key) or not. Here, values can be stored at both the leaf nodes and internal nodes.

The algorithm works on the basis of common prefixes. The path from the root to a node encodes the prefix. Consider two keys "travel" and "truck". In this case, we store the common prefix "tr" at the root node and add two children to the root node: 'a' and 'u', respectively. We proceed similarly and continue to create common prefix nodes across keys. Consider two more keys "tram" and "tractor". In this case, after we traverse the path with the prefix "tra", we create two leaf nodes "ctor" and "m". If we were to now add a new key "trams", then we would need to create a new child "s" with the parent as the erstwhile leaf node labeled "m". In this case, both "tram" and "trams" would be valid keys. Hence, there is a need to annotate every internal node with an extra bit to indicate that the path leading from the root to that node corresponds to a valid key. We can associate a value with any node that has a valid key. In other words, this would mean that the path from the root to the leaf node corresponds to a valid key.

The advantage of such a structure is that we can store a lot of keys very efficiently and the time it takes to traverse it is proportional to the number of letters within the key. Of course, this structure works well when the keys share reasonably long prefixes. Otherwise, the tree structure will not form, and we will simply have a lot of separate paths. Hence, whenever there is a fair amount of overlap in the prefixes, a radix tree should be used. It is important to understand that the lookup time complexity is independent of the number of keys – it is theoretically only dependent on the number of letters (digits) within a key.

Insertion and deletion are easy. We need to first perform a lookup operation and find the point at which the non-matching part of the current key needs to be added. There is a need to add a new node that branches out of an existing

node. Deletion follows the reverse process. We locate the key first, delete the node that stores the suffix of the string that is unique to the key and then possibly merge nodes.

There is a popular data structure known as a *trie*, which is a prefix tree like a radix tree with one important difference: in a trie, we proceed letter by letter. This means that each edge corresponds to a single letter. Consider a system with two keys "tractor" and "tram". In this case, we will have the root node, an edge corresponding to 't', then an edge corresponding to 'r', an edge corresponding to 'a', so on and so forth. There is no point in having a node with a single child. We can compress this information to create a more efficient data structure, which is precisely a radix tree. In a radix tree, we can have multi-letter edges. In this case, we can have an edge labeled "tra" (fuse all single-child nodes).

C.5.1   Patricia Trie

A Patricia trie or a Patricia tree is a special variant of a radix tree, where all the letters are binary (0 or 1). Similar to a radix tree, it is a compressed data structure. We do not have an edge for every single binary bit in the key. Instead, we have edges labeled with multiple bits such that the number of internal nodes is minimized. Assume a system with only two keys that are not equal. Regardless of the Hamming distance between the two keys, the Patricia Trie will always have three nodes – a root and two children. The root node will store the shared prefix, and the two children will contain the non-shared suffix of the binary keys. Incidentally, *Patricia* stands for Practical Algorithm To Retrieve Information Coded In Alphanumeric.

C.6   Augmented Tree
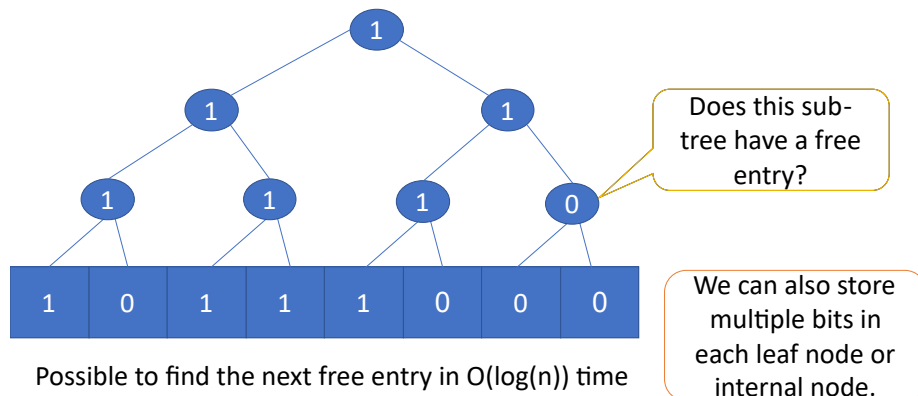
Figure C.3: Example of an augmented tree

This kind of data structure is very useful for representing information stored in a bit vector.

Let us elaborate. Assume a very long vector of bits. This is a reasonably common data structure in the kernel particularly when we consider page allocation. Assume a system that has a million frames (physical pages) in the physical address space, and we need to manage this information. We can represent this with a bit vector that has a million 1 bit-sized entries. If the value of the ith entry is 1, then it means that the corresponding physical page is free, and the value 0 means that the corresponding physical page has been allocated.

Now a common operation is to find the first physical page that has not been allocated such that it can be allocated to a new process. In this case, we need to find the location of the first 1 in the bit vector starting from a given position and proceeding towards the right. On the same lines, we can have an analogous problem where the task is to find the first 0 in the bit vector. Regardless of whether we are searching for a 0 or 1, we need a data structure to locate such positions efficiently.

A naive algorithm is to of course start traversing the bit vector from the lowest address to the highest address and terminate the search whenever a 0 or 1 is found. If we reach the end of the bit vector and do not find the entry of interest, then we can conclude that no such entry exists. Now, if there are n entries, then this algorithm will take $O(n)$ time, which is too slow. We clearly need a much faster algorithm, especially something that runs in $O(log(n))$ time.

This is where an augmented tree is very useful. We show an example in Figure C.3. We treat the single-bit cells of the bit vector as leaf nodes. Adjacent cells have a parent node in the augmented tree. This means that if we have n entries in the bit vector, then there are $n/2$ entries in the second last level of the tree. This process continues towards the root (in a similar fashion). We keep on grouping adjacent internal nodes, and create a parent for them until we reach the root. We thus end up with a balanced binary tree if n is a power of 2. The greatness of the augmented tree lies in the contents of the internal nodes. To explain this, let us start with the root. Assume that we are searching for the next position that stores a '1'.

If the root node stores a 1, it means that at least a single location in the bit vector stores a 1. This is a very convenient trick because we instantly know if the bit vector contains all 0s, or it has at least one position that stores the bit 1. Each of its children is the root of a subtree (contiguous region in the bit vector). It stores exactly the same information as the root. If the root of the subtree stores a 0, then it means that all the bit vector locations corresponding to the subtree store a 0. If it stores a 1, then it means that at least one location stores a 1.

Now let us consider the problem of locating the first 1 starting from the lowest address (from the left in the figure). We first check the root. If it contains a 1, then it means that there is at least a single 1 in the bit vector. We then proceed to look at the left child. If it contains a 1, then it means that the first half of the bit vector contains a 1. Otherwise, we need to look at the right child of the root. This process continues recursively until we reach the leaf nodes. At each stage we prefer the left child over the right child. We are ultimately guaranteed to find a position that contains 1 if the root contains 1 (there is an entry in the bit vector that contains it).

This is a very fast process and runs in logarithmic time. Whenever we change a value from $0 \rightarrow 1$ in the bit vector, we need to walk up the tree and convert all 0s to 1 on the path. However, when we change a value from $1 \rightarrow 0$, it is slightly

tricky. We need to traverse the tree towards the root, however we cannot blindly convert 1s to 0s. Whenever, we reach a node on the path from a leaf to the root, we need to take a look at the contents of the other child and decide accordingly. If the other child contains a 1, then the process terminates right there. This is because the parent node is the root of a subtree that contains a 1 (via the other child). If the other child contains a 0, then the parent's value needs to be set to 0 as well. This process terminates when we reach the root.

C.6.1   Bloom Filters

A Bloom filter is used to check for set membership. It answers queries of the kind, "Is element x a member of set $\mathcal{S}$?" Its operating principle is quite simple – it is an extension of hashing. Before we proceed further, we need to note that it is a probabilistic data structure. There can be false positives but no false negatives. This means that if an answer to a membership query is in the affirmative, then the element may be present or may not be present. However, if the answer is negative, then the element is not present in the set for sure.
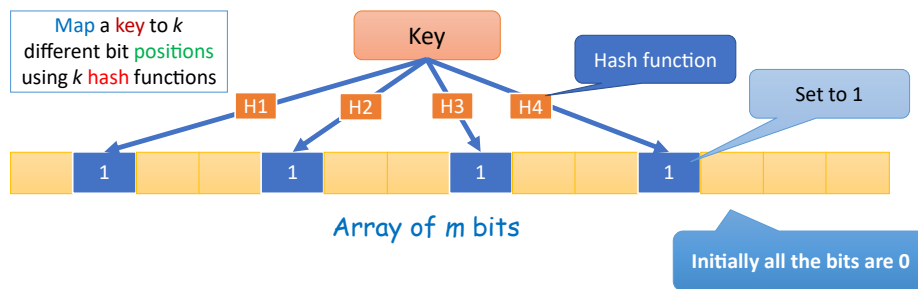
Figure C.4: A Bloom filter

This is achieved as follows (refer to Figure C.4). A key is associated with k different hash functions. Each hash function maps the key to a position in a large array that contains m bits. While adding a key, we just set the bits at all the mapped bit positions to 1 as shown in the figure. Note that all the elements of the bit vector are initialized to 0.

While searching for a key, we compute the values of the k different hash functions. Next, we inspect the bits at all the k corresponding bit positions. If all of them are 1, then the key may be present in the set. The reason we use the phrase "may be" is because it is possible that half the bits were set because of key x and the rest half were set because of another key y. It is not possible to find out if this is indeed the case. Hence, the answer that we get in this case is a probabilistic "Yes".

However, when one of the bits is 0, we can be sure that the associated key is definitely not present. If it is actually present, all the bits would have been 1 for sure. We shall find very interesting uses for such data structures in the kernel.

Note that such a data structure has numerous shortcomings. We cannot delete entries. Naively setting the k bits associated with a key to 0 will not work. It is possible that there are multiple keys that map to a subset of these

bits. All of them will get removed, which is something that we clearly do not want. One option is to store a counter at each entry instead of a bit. When a key is added to the set, we just increment all the associated counters. This is fine as long as we do not have overflows. One of the important reasons for opting for a Bloom filter is its simplicity and compactness. This advantage will be lost if we start storing large counters in each entry. With this approach removing a key is very easy – we just decrement the associated counters. Nevertheless, the overheads can be sizable and the benefits of compactness will be lost. Hence, counters are normally not used in Bloom filters.

The other issue is that bits get flipped in only one direction, 0 to 1. They never get flipped back because we do not do anything when an element is removed. As a result, the Bloom filter becomes full of 1s with the passage of time. There is thus a need to periodically reset the bits.

Bibliography

[Belady et al., 1969] Belady, L. A., Nelson, R. A., and Shedler, G. S. (1969). An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of the ACM*, 12(6):349–353.

[Community,] Community, K. D. Rcu concepts. Online. Available at: `https://docs.kernel.org/RCU/index.html`.

[Corbet, 2010] Corbet, J. (2010). The case of the overly anonymous anon_vma. Online. Available at: `https://lwn.net/Articles/383162/`.

[Corbet, 2014] Corbet, J. (2014). Locking and pinning. Online. Available at: `https://lwn.net/Articles/600502/`.

[Corbet, 2021] Corbet, J. (2021). Clarifying memory management with page folios. Online. Available at: `https://lwn.net/Articles/849538/`.

[Corbet, 2022] Corbet, J. (2022). A memory-folio update. Online. Available at: `https://lwn.net/Articles/893512/`.

[Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, third edition.

[Corporation, 2024a] Corporation, I. (2024a). *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide*. Consists of Volumes 3A, 3B, 3C, and 3D.

[Corporation, 2024b] Corporation, I. (2024b). *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 4: Model-Specific Registers*.

[de Olivera, 2018] de Olivera, D. B. (2018). Avoid __schedule() being called twice, the second in vain. Online. Available at: `https://www.mail-archive.com/linux-kernel@vger.kernel.org/msg1740572.html`.

[Fornai and Iványi, 2010a] Fornai, P. and Iványi, A. (2010a). Fifo anomaly is unbounded. *Acta Univ. Sapientiae*, 2(1):80–89.

[Fornai and Iványi, 2010b] Fornai, P. and Iványi, A. (2010b). Fifo anomaly is unbounded. *arXiv preprint arXiv:1003.1336*.

[Graham, 1969] Graham, R. L. (1969). Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429.

[Herlihy and Shavit, 2012] Herlihy, M. and Shavit, N. (2012). *The Art of Multiprocessor Programming*. Elsevier.

[Howlett, 2021] Howlett, L. (2021). The maple tree, a modern data structure for a complex problem. Online. Available at: `https://blogs.oracle.com/linux/post/the-maple-tree-a-modern-data-structure-for-a-complex-problem`.

[Karger et al., 1999] Karger, D. R., Stein, C., and Wein, J. (1999). Scheduling algorithms. *Algorithms and theory of computation handbook*, 1:20–20.

[Lameter and Kumar, 2014] Lameter, C. and Kumar, P. (2014). this_cpu operations. Online. Available at: `https://docs.kernel.org/core-api/this_cpu_ops.html`.

[Lehoczky, 1990] Lehoczky, J. P. (1990). Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *[1990] Proceedings 11th Real-Time Systems Symposium*, pages 201–209. IEEE.

[License, 1989] License, G. G. P. (1989). Gnu general public license, version 2.

[Mall, 2009] Mall, R. (2009). *Real-time systems: theory and practice*. Pearson Education India.

[McKenney, 2003] McKenney, P. (2003). Reader-writer locking/rcu analogy. Online. Available at: `https://www.usenix.org/legacy/publications/library/proceedings/usenix03/tech/freenix03/full_papers/arcangeli/arcangeli_html/node7.html`.

[McKenney, 2008] McKenney, P. (2008). Hierarchical rcu. Online. Available at: `https://lwn.net/Articles/305782`.

[McKenney, 2007] McKenney, P. E. (2007). What is rcu, fundamentally? Online. Available at: `https://lwn.net/Articles/262464/`.

[Molnar, 2006] Molnar, I. (2006). Runtime locking correctness validator. Online. Available at: `https://www.kernel.org/doc/html/latest/locking/lockdep-design.html`.

[Rapoport, 2019] Rapoport, M. (2019). Memory: the flat, the discontiguous, and the sparse. Online. Available at: `https://lwn.net/Articles/789304/`.

[Rybczynska, 2021] Rybczynska, M. (2021). Introducing maple trees. Online. Available at: `https://lwn.net/Articles/845507/`.

[Sarangi, 2021] Sarangi, S. R. (2021). *Basic Computer Architecture*. White Falcon Publishing, 1st edition edition.

[Sarangi, 2023] Sarangi, S. R. (2023). *Next-Gen Computer Architecture*. White Falcon, 1st edition edition.

[Singh and Sarangi, 2020] Singh, S. S. and Sarangi, S. R. (2020). Softmon: A tool to compare similar open-source software from a performance perspective. In *Proceedings of the 17th International Conference on Mining Software Repositories*, page 397–408.

[Zimmer et al., 2021] Zimmer, V., Banik, S., and Regupathy, R. (2021). Early platform hardening technology for slimmer and faster boot. US Patent App. 17/109,081.

Index