Operating Systems: A Linux Kernel-Oriented Approach

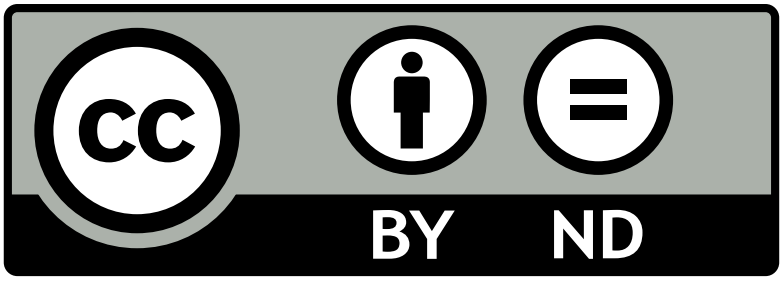
(Partially written.
Updates are released every week on Fridays.)

Send bug reports/suggestions to
srsarangi@cse.iitd.ac.in

Version 0.51

Smruti R. Sarangi

April 29, 2024



List of Trademarks

- Microsoft and Windows are registered trademarks of Microsoft Corporation.

Contents

1 Introduction **7**
- 1.1 Types of Operating Systems . . . 9
- 1.2 The Linux OS . . . 10
  - 1.2.1 Versions, Statistics and Conventions . . . 11

2 Basics of Computer Architecture **15**
- 2.1 Cores, Registers and Interrupts . . . 15
  - 2.1.1 Multicore Systems . . . 15
  - 2.1.2 Inside a Core . . . 16
  - 2.1.3 Registers . . . 17
  - 2.1.4 Interrupts, Exceptions, System Calls and Signals . . . 19
- 2.2 Virtual Memory . . . 24
  - 2.2.1 Memory Map of a Process . . . 27
  - 2.2.2 Idea of Virtual Memory . . . 28
  - 2.2.3 Implementation of the Address Translation System . . . 32
  - 2.2.4 Segmented Memory . . . 38
- 2.3 I/O System . . . 40
  - 2.3.1 Overview . . . 41
  - 2.3.2 I/O via I/O Ports . . . 42
  - 2.3.3 Memory-Mapped I/O . . . 43
  - 2.3.4 I/O using DMA . . . 44

3 Processes **47**
- 3.1 The Notion of a Process . . . 48
- 3.2 The Process Descriptor . . . 48
  - 3.2.1 struct task_struct . . . 48
  - 3.2.2 `struct thread_info` . . . 49
  - 3.2.3 Task States . . . 51
  - 3.2.4 Kernel Stack . . . 53
  - 3.2.5 Task Priorities . . . 55
  - 3.2.6 Computing Actual Task Priorities . . . 56
  - 3.2.7 `sched_info` . . . 57
  - 3.2.8 Memory Management . . . 58
  - 3.2.9 Storing Virtual Memory Regions . . . 59
  - 3.2.10 The Process Id . . . 60
  - 3.2.11 Processes and Namespaces . . . 61
  - 3.2.12 File System, I/O and Debugging Fields . . . 66
  - 3.2.13 The PTrace Mechanism . . . 67

3.3 Process Creation and Destruction . . . 67
3.3.1 The Fork Mechanism . . . 68
3.3.2 The exec Family of System Calls . . . 73
3.3.3 Kernel Threads . . . 74
3.4 Context Switching . . . 75
3.4.1 Hardware Context . . . 75
3.4.2 Types of Context Switches . . . 76
3.4.3 Details of the Context Switch Process . . . 78
3.4.4 The Context Switch Process . . . 80

4 System Calls, Interrupts, Exceptions and Signals 83
4.1 System Calls . . . 84
4.1.1 General Concepts about Underlying Design . . . 85
4.1.2 The OS Side of Things . . . 85
4.1.3 Returning from a System Call . . . 86
4.2 Interrupts and Exceptions . . . 87
4.2.1 APICs . . . 88
4.2.2 IRQs . . . 90
4.2.3 Kernel Code for Interrupt Handling . . . 93
4.2.4 Managing the IDT Table . . . 94
4.2.5 The Interrupt Path . . . 96
4.2.6 Exceptions . . . 97
4.3 Softirqs, Threaded IRQs, Work Queues . . . 101
4.3.1 Softirqs . . . 102
4.3.2 Threaded IRQs . . . 103
4.3.3 Work Queues . . . 104
4.4 Signal Handlers . . . 108
4.4.1 Example of a Signal Handler . . . 108
4.4.2 Signal Delivery . . . 109
4.4.3 Kernel Code . . . 112
4.4.4 Entering and Returning from a Signal Handler . . . 116

5 Synchronization and Scheduling 119
5.1 Synchronization . . . 119
5.1.1 Introduction to Data Races . . . 119
5.1.2 Design of a Simple Lock . . . 122
5.1.3 Theory of Data Races . . . 124
5.1.4 Deadlocks . . . 126
5.1.5 Pthreads and Synchronization Primitives . . . 130
5.1.6 Theory of Concurrent Non-Blocking Algorithms . . . 134
5.1.7 Progress Guarantees . . . 139
5.1.8 Semaphores . . . 142
5.1.9 Condition Variables . . . 143
5.1.10 Reader-Writer Lock . . . 144
5.1.11 Barriers and Phasers . . . 145
5.2 Queues . . . 147
5.2.1 Wait-Free Queue . . . 148
5.2.2 Queue with Mutexes . . . 150
5.2.3 Queue with Semaphores . . . 151
5.2.4 Queue with Semaphores but No Busy Waiting . . . 151

5.2.5 Reader-Writer Lock . . . 153
5.3 Concurrency within the Kernel . . . 154
5.3.1 Kernel-Level Locking: Spinlocks . . . 155
5.3.2 Kernel Mutexes . . . 160
5.3.3 Kernel Semaphores . . . 162
5.3.4 The lockdep Mechanism . . . 162
5.3.5 The RCU Mechanism . . . 164
5.4 Scheduling . . . 165
5.4.1 Space of Scheduling Problems . . . 165
5.4.2 Single Core Scheduling . . . 167
5.4.3 Multicore Scheduling . . . 172
5.4.4 Banker's Algorithm . . . 175
5.4.5 Scheduling in the Linux Kernel . . . 180
5.4.6 Completely Fair Scheduling (CFS) . . . 185
5.4.7 Real-Time and Deadline Scheduling . . . 190

6 The Memory System 191
6.1 Traditional Heuristics for Page Allocation . . . 191
6.1.1 Base-Limit Scheme . . . 191
6.1.2 Classical Schemes to Manage Virtual Memory . . . 193
6.1.3 The Notion of the Working Set . . . 201
6.2 Virtual and Physical Address Spaces . . . 203
6.2.1 The Virtual Memory Map . . . 203
6.2.2 The Page Table . . . 205
6.2.3 Pages and Folios . . . 208
6.2.4 Managing the TLB . . . 211
6.2.5 Partitioning Physical Memory . . . 214
6.3 Page Management in the Kernel . . . 221
6.4 Kernel Memory Allocation . . . 222
6.4.1 Buddy Allocator . . . 222
6.4.2 Slab Allocator . . . 227
6.4.3 Slub Allocator . . . 229

7 The I/O System and Device Drivers 231

8 Virtualization and Security 233

A The X86-64 Assembly Language 235
A.1 Registers . . . 235
A.2 Basic Instructions . . . 238

B Compiling, Linking and Loading 241
B.1 The Process of Compilation . . . 241
B.1.1 Compiler Passes . . . 242
B.1.2 Dealing with Multiple C Files . . . 243
B.1.3 The Concept of the Header File . . . 244
B.2 The Linker . . . 246
B.2.1 Static Linking . . . 247
B.2.2 Dynamic Linking . . . 250
B.2.3 The ELF Format . . . 252

B.3 Loader 253

C Data Structures **255**
C.1 Linked Lists in Linux 255
C.1.1 Singly-Linked Lists 257
C.2 Red-Black Tree 258
C.3 B-Tree 259
C.3.1 The Search Operation 259
C.3.2 The Insert and Delete Operations 260
C.3.3 B+ Tree 261
C.4 Maple Tree 261
C.5 Radix Tree 261
C.5.1 Patricia Trie 262
C.6 Van Emde Boas Tree 263

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 (to some extent). In this case, it is a special program, which exercises some control over hardware and other programs. In other words, it has special features and facilities to manage all aspects of the underlying hardware and to also ensure that a convenient interface is provided to high-level application software such that they can seamlessly operate oblivious to the idiosyncrasies of the underlying hardware. The basic question that needs to be answered is, "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 also should not be a goal. We do not want normal programs to have access to the 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 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 have the underlying hardware, high-level programs running on top of it and the OS sits in the middle. It acts as a mediator, a broker, a security manager and an overall resource manager. Let us thus quickly summarize the need for having a special program like an OS.

- 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 al-

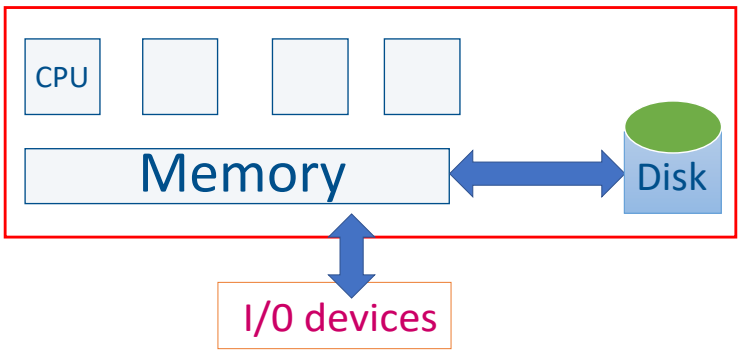


Figure 1.1: Diagram of the overall system

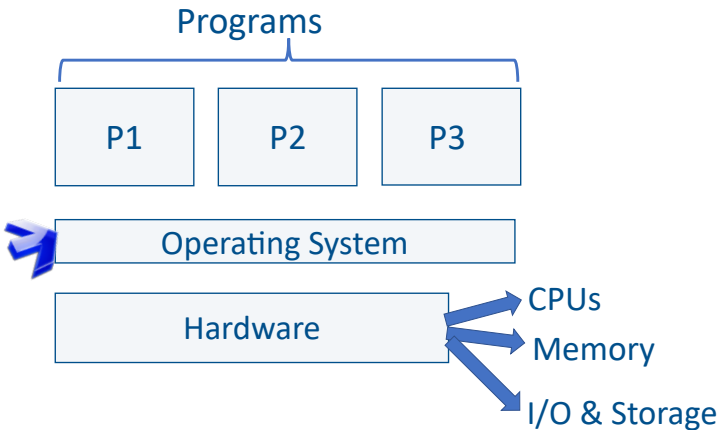


Figure 1.2: Place of the OS in the overall system

location with predefined priorities in mind.

- 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 such concurrent accesses are disciplined and it is not possible for one resource to be used in parallel by multiple 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 security violation or this may be a genuine shared memory-based

communication pattern. There is a need to differentiate one from the other by providing neat and well-defined mechanisms.

- Power, temperature and security concerns have become very important over the last decade. The reason being that any operating system that is being designed today also needs to run on very small devices such as mobile phones, tablets and even smart watches. In the foreseeable future, they may run on even smaller devices such as smart glasses or may even be embedded within the body. Hence, it is important for an OS to make it possible for the administrator to configure different power policies, which it must strictly adhere to.

Definition 1
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 in a secure fashion 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. And clearly, for such situations, the chances of viruses or malware attacks are elevated. Hence, security is a first-order concern.

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. However, 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 smart watches. 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. Furthermore,

in this case the port size and the memory footprint of the OS needs to be very small.

1.2 The Linux OS

In this course, will be teaching generic OS concepts in the context of the Linux OS. As compared to all other operating systems, Linux has a very different history. It is not been written by one particular person or one particular organization. In fact, it is a modern marvel in the sense that it has been written via 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 now has widespread acceptance in all kinds of computing platforms ranging from smart watches to laptops to high-end servers.

It all started 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 little bit of code and some of them contributed in other ways as well 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) umbrella, a lot of software, specifically utilities, were being developed. The Linux developers found a common cause with the developers in the free software movement and were able to incorporate many of the utilities that they developed into the Linux operating system. This was a good fusion of communities, which led to rapid development of the new OS.

Way back in 1992, 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 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/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. This was explicitly forbidden. This ensured that whatever changes and modifications are made to any piece of code that comes with a GPL license, all the changes are visible to the community and thus they can build on this. There were no proprietary walls; this allowed the community to make rapid progress. 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.

Over the years, Linux grew 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. Note that Linux was reasonably similar to UNIX in terms of the interface and some high-level design decisions. The year 2003 was a pivotal year for Linux, because this year, Linux kernel version 2.6 was released. It had a lot of changes and was very different from the previous kernel versions. After this, Linux started being taken very seriously in both academic and enterprise circles. In a certain sense, it 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. Red Hat(®), Suse(®)and Ubuntu(®)(Canonical(®)) where some of the major vendors that dominated the scene. As of writing this book, circa 2023, these 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 Linux. Many of the operating systems for smart devices and other wearable gadgets are also based on Android. In addition, Google(®)'s Chrome OS also is a Linux-derived variant. So are other operating systems for Smart TVs such as LG(®)'s webOS and Samsung(®)'s Tizen.

As of today, Linux is not the only free open source operating system. There are many others, which are derived from classical UNIX, notably the Berkeley standard distribution (BSD) variant. Some of the important variants are FreeBSD, OpenBSD and NetBSD. Similar to Linux, their code is also Three to use and distribute. Of course, they follow a different licensing mechanism. 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 puts them to new hardware. The paper by Singh et al. [Singh and Sarangi, 2020] nicely compares the three operating systems in terms of performance for different workloads (circa 2015).

1.2.1 Versions, Statistics and Conventions

In this book, we will be primarily teaching generate OS concepts. however, every operating system concepts needs to be explained in the light of a solid practical implementation. This is where the latest version of the current as of 2023 will be used. Specifically we will use kernel version v6.2 explain OS concepts. All the OS code that we shall show will be from the main terminal branch. It is available at `https://elixir.bootlin.com/linux/v6.2.16/source/kernel`.

Let us now go through some of the key statistics vis-a-vis The Linux kernel version v6.2 has roughly 25 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.

Consider a Linux version 6.2.16. Here 6 is the major version, 2 is the minor version number and 16 is the patch number. Every $\langle major, minor \rangle$ version pair has multiple patch numbers associated with it. Every time there is a major commit, a patch is created. If a minor version number is even, then it means that it is a production version. This version can be released to the public and can be used for regular applications (desktop or enterprise). Odd numbers are development versions where new features and the effects of bug fixes are

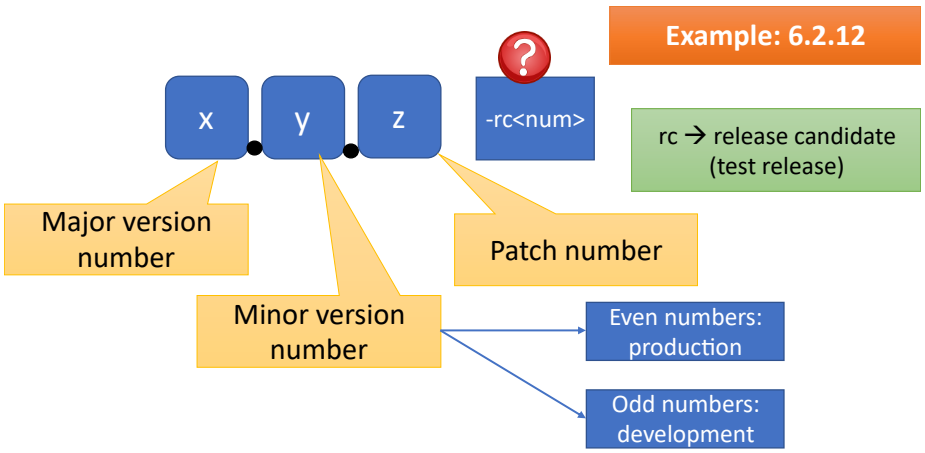


Figure 1.3: Logic for assigning Linux versions

incorporated. Such a patch is considered to be *stable*. It is a non-trivial feature addition and is considered to be stable.

Each such patch is associated with multiple release candidates. A release candidate does not have major bugs; however, it incorporates multiple smaller fixes and feature additions that are not fully verified. These release candidates are still considered experimental and are not fully ready to be used in any kind of a production-level setting. They are numbered as follows -rc1, -rc2, ... They are mainly aimed at other Linux developers. They 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 can be succeeded by a stable version (read patch or major/minor version).

Details 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 functions. 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 1.5 million lines of code.

The other large subsystems that contain large volumes of code are the directories for the filesystem and network. Note that a popular OS such as Linux need to support a large number of file systems and network protocols. As a

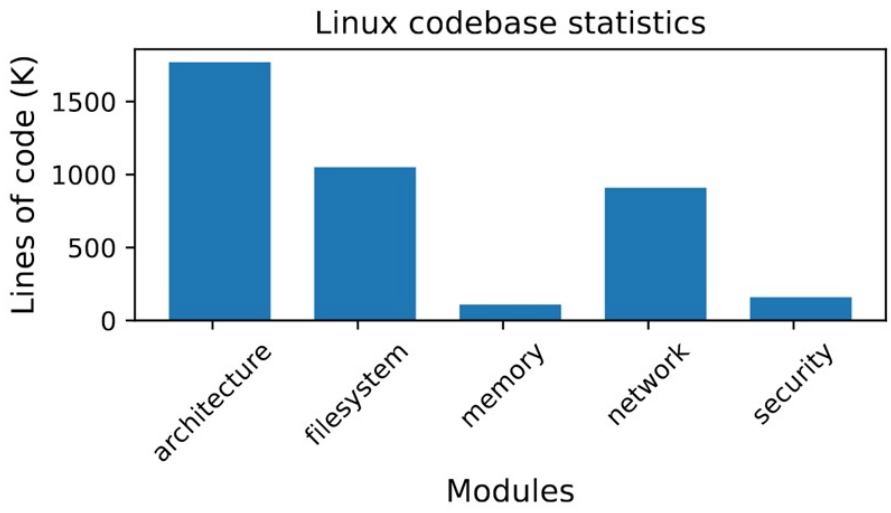


Figure 1.4: Breakup of the Linux code base

result, the code base for these directories is quite large. The other subsystems for the memory and security modules are comparatively much smaller.

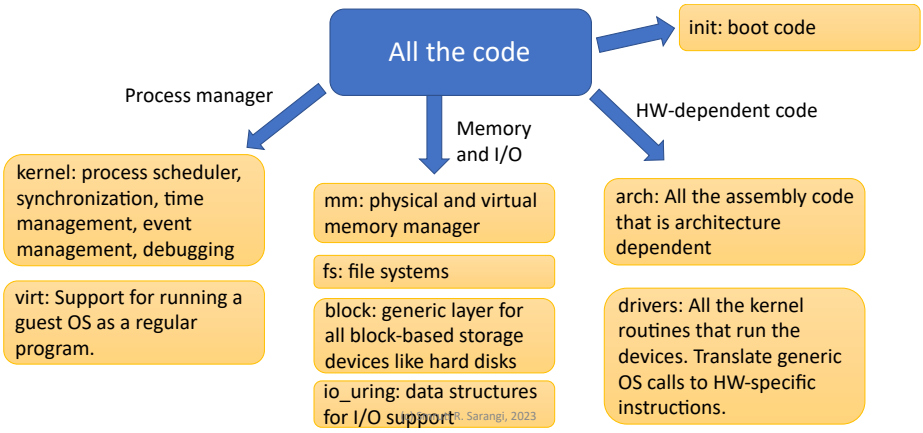


Figure 1.5: Important directories in the Linux kernel code base

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 or the core 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. These modules are related to the code to virtualize the operating system *virt*, where we can run the OS as a regular program on top of another OS. The virtualization subsystem is tightly coupled with the memory, file and I/O subsystems .

Finally, the largest directory is *drivers* that contains drivers (specialized programs for talk to devices) for a host of I/O devices. Of course, there are multiple things to be considered here. We don't want to include the code of every single device on the planet in the code base of the kernel. It will become too large. At the same time, there is an advantage. The kernel can seamlessly run on a variety of hardware. Hence, the developers of the kernel need to wisely choose the set of drivers that they need to include in the code base that is released and distributed. The devices should be very popular and the drivers should be deemed to be safe (devoid of security issues).

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. 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 can recapitulate all the important concepts and understand some of the key hardware features that modern OSes rely on.

2.1 Cores, Registers and Interrupts

2.1.1 Multicore Systems

Figure 2.1 shows an overview of a typical multicore processor. As of 2023, 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 of the very recent processors are including 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 2023) and needless to say the slowest of all. It is typically made up of DRAM memory cells that are slow yet have a very high storage density. The most

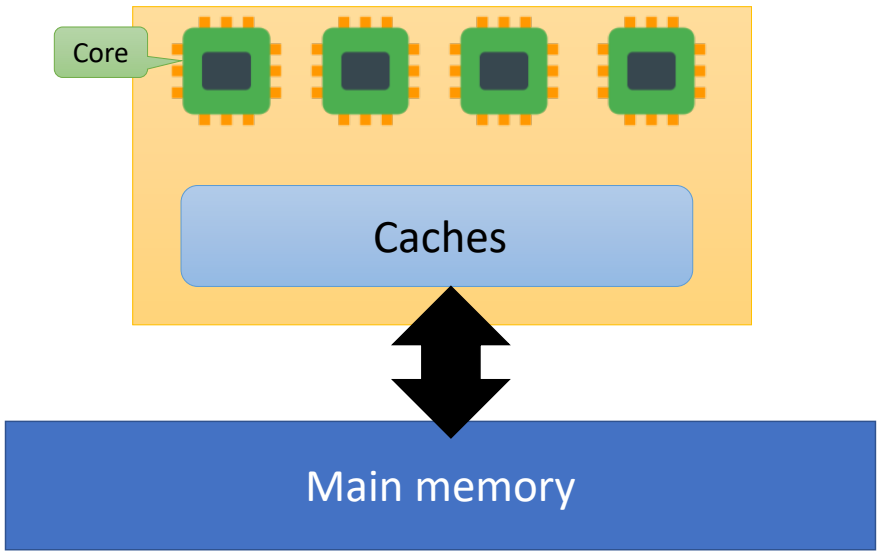


Figure 2.1: A multicore processor

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 memories inside the chip such as the L1, L2 and L3 caches are 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.

A software program including the OS perceive the 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. Of course, later on when we discuss virtual memory, we will have to change this abstraction. But, even then, large parts of this abstraction will continue to hold.

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: L1-level instruction and data caches. The core also has a set of named storage locations that are accessible by instructions directly known as *registers*. A typical processor has 8-32 registers. The advantage of having these 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. Register File

Most of the operations in a core happen on the registers. Registers are often

both the operands and even when a location in memory needs to be accessed, it is accessed using 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 that is stored in the %esp register plus 4. As we can see, registers are ubiquitous. They are used to access memory addresses and as we shall see later, I/O address as well. Let us differentiate between CISC and RISC processors here. RISC processors tend to use registers much more than CISC processors that have a fair number of memory operands in their instructions. In any case, registers are central to the operation of any program (be it RISC or CISC), and needless to say the compiler needs to be aware of them.

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*. These are visible to all software including the compiler. Note that almost all the programs that are compiled today will 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 can see. In Chapter 8, we will also look at hypervisors or virtual machine managers (VMMs) that also 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 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 access these registers to enable multi-tasking: 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 turn of the system itself. We do not want all of these instructions to be visible to regular programs because then a single application can create havoc. Because of this, we entrust only a specific set of programs (OS and VMM) with access to these registers.

There are debug registers that are meant to debug hardware and system software. Given the fact that they are privy to more 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 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 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, network card, etc. Here again, we need privileged access. Otherwise, we can have serious security violations

and different applications may try to monopolize an I/O resource and not allow other applications to access them. Hence, the OS needs to act as a broker and this is possible only why access to the device needs to be restricted.

Given the fact that we have discussed so much about privileged registers, we should now see how the notion of privileges is implemented and how we 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, we 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 enable 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 of one indicated that an application program is executing.

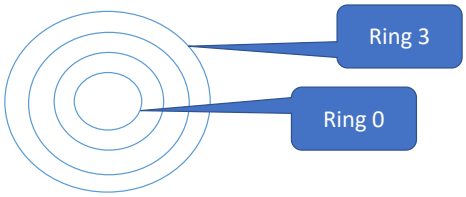


Figure 2.2: Rings in x86 processors

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 2.2). The primary role of rings 1 and 2 is to run guest operating systems and other software that do not require as much of privileged access as the software running in ring zero. Nevertheless, they will have enjoy more privileges than regular application code. They are typically used while running guest OSes on virtual machines.

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 at a lower privilege level (higher ring) accesses instructions that should be executed by code at a higher privilege level (lower ring)? In general, we would expect that there will be an exception. Then the appropriate exception handler will 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 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 complicated. The main reason for this is that when instructions were created initially while designing the ISA, virtual machines were not around and thus designers could not think that far enough. 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.

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 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 some kind of 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 a 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 directly be invoked by the application. Hence there is a need to generate a dummy interrupt such that The same set of actions can take place, which happens

when an external interrupt is received. 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, an 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 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 distributions of Linux.

In the header file `/usr/include/asm/unistd_64.h`, 286 System calls are defined. The standard way to call 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. We generate a dummy interrupt, store some data corresponding to the state of the executing program (for more details, refer to [Sarangi, 2021]) and load 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 this needs 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 back to exactly the same point at which the context was switched.

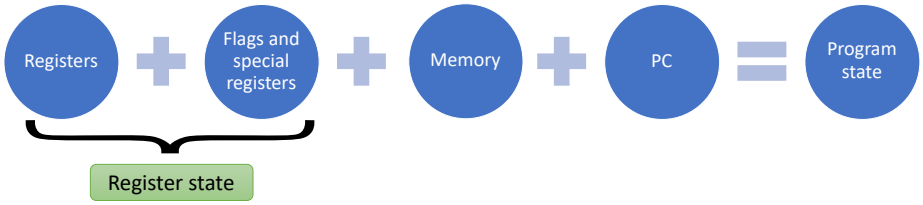


Figure 2.3: The "context save" process

Figure 2.3 show 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 state very effectively. Hence, we need not bother about storing and restoring the memory state because there is already a mechanism namely virtual memory that takes care of it completely. 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/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. Note that for the sake of readability, we will use the term interrupt handler to refer to a traditional interrupt handler as well as exception handlers and system call handlers wherever this is clear from the

context.

Subsequently, the first task of the interrupt handler is to retrieve the program state 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 be more work and retrieve the full program state. In any case, 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 2.4. 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 an exception.

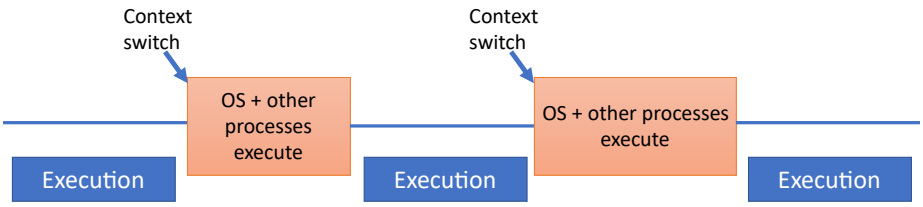


Figure 2.4: The life cycle of a process (active and interrupted phases)

We can visualize 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.

Timer Interrupts

There is an important question to think about here. What if there is a program that does not see any interrupts and there are no system calls or exceptions? This means that the OS will never get executed if all the cores are occupied by different instances of such programs. Kindly note that the operating system is never executing in the background (as one would want to naively believe) – it is a *separate program* that needs to be invoked by a very special method namely either a system call, exception or interrupt. Let us refer to system calls, exceptions and interrupts as *events of interest*. It 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 and if these programs decide to run

for a long time, is rebooting the system the only option?

Question 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 cannot always rely on system calls, exceptions and interrupts (events of interest) to bring in the operating system. This is because as we have shown in question 1, 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.

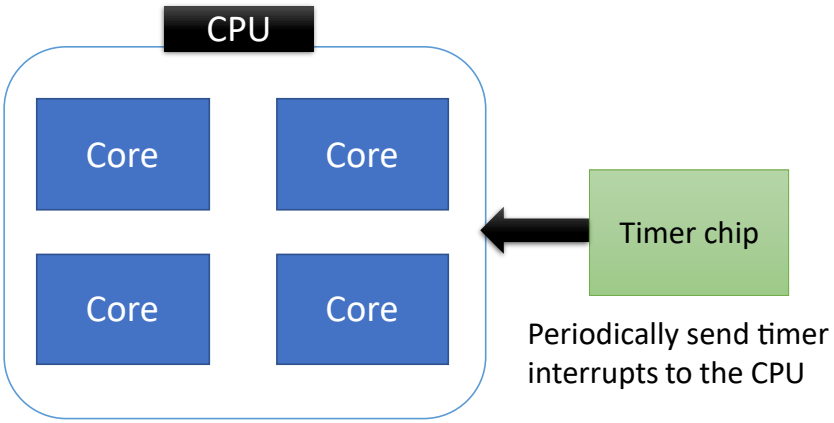


Figure 2.5: The timer chip generates periodic interrupts

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 2.5, 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 source of dummy 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 most integral part of the machine that supports an operating system. The key insight is that this 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*, where there is a timer interrupt at the end of every *jiffy*. The number of jiffys (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 seen, the OS gets invoked on one core and now its job is to take control of the system and basically manage everything including 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 always chooses the most eligible process (using its scheduler) and runs it on a core.

If the new process runs 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. However, if a process on some other core needs to be swapped out, and it needs to be replaced with the chosen process, then the process is more elaborate. It is necessary to send an interrupt to that core such that the OS starts running on that core. There is a mechanism to do so – it is called 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 uses such APIs to run the OS on a given core. The OS may choose to do more of management and bookkeeping activities or quickly find the next process to run and run it on that core.

2.2 Virtual Memory

The first question that we would like to ask is 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.

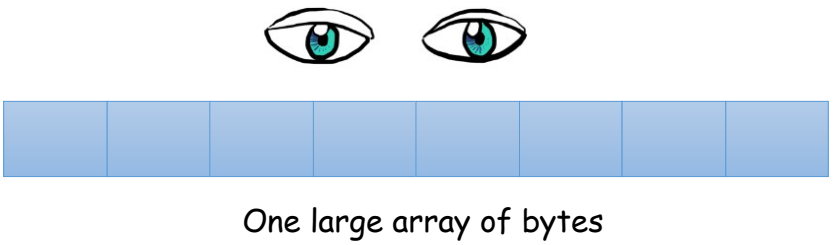


Figure 2.6: The way that a programmer or compiler view the memory space

Such a simple abstraction is shown in Figure 2.6. 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. If we think about it, we cannot make any other assumption; this is the most elegant and simplifies the job of everybody other than the engineers designing the memory system. The complexity of the memory system stems from the fact that in any ISA, load and store instructions primarily read their addresses from registers. They can add a constant offset to it, however, the fact still remains that most of the address information comes from registers. The size of a register thus determines the range of addresses that can be accessed.

The Compatibility Problem

This is why in an n-bit architecture, where the register size is n bits, the most straightforward thing to assume is that the instructions can access any of the addressable 2^n bytes. A similar assumption needs to be made by the programmer and the compiler because they only see 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, no other assumption can be made. This may sound impractical on 64-bit machines, but this is the most elegant assumption that can be made. Of course, how we design a memory system whose size is much smaller than 2^{64} bytes remains a problem to be solved.

What if we made another assumption? If we assumed that the program can access 2 GB at will, it will not run on a system with 1 GB of memory, unless we find a mechanism to do so. If we can find a mechanism to do this, then

we can always scale the system to assume that an address is 2^{32} or 2^{64} bytes and still manage to run on physical systems with far lower memory. We thus have a *compatibility problem* here, where we want our program to assume that addresses are n bits wide (typically 32 or 64 bits), yet run on machines with all memory sizes (typically much lower than the theoretical maximum).

Definition 2 *Processes assume that they can access any byte in large memory regions 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 fact is that the addresses they assume are not compatible with physical addresses (on 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. Then when the 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.

The Overlap Problem

In this case, unbeknownst to both the 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 process'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 overlap between the sets. This is known as the *overlap problem.*

Definition 3 *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.

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 2.7 shows the memory map of a process on the Linux operating system. The memory map is a layout of the memory space that a process can access. It shows where each type of data (or code) is stored.

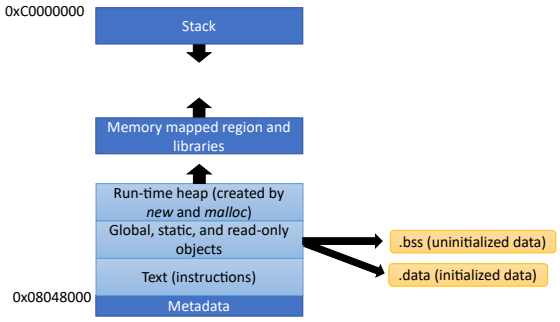


Figure 2.7: The memory map of a process in 32-bit Linux

What we can see is that the memory map is partitioned into distinct zones. The memory map 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. Note that each of these sections in the memory map is basically a range of memory addresses and this range varies from process to process. 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 on a 32-bit system, an application is given 3 GB of memory at most. This is why the stack section starts at this point. Of course, 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 we typically use a 64-bit machine where the likelihood of this happening is very low because our programs are not that large at the moment.

The last unanswered question is what happens to the one GB 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 regards to the kernel memory is that it is an invariant across process memory maps. It is something that pretty much remains constant and in the case of a 32-bit system, occupies the top one GB of the memory map of every process. In a certain sense, processors assume that their range of operation is the first 3 GB and the top one GB is beyond their jurisdiction.

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 correlated 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. Additionally, this problem creates a security nightmare.

2.2.2 Idea of 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 many processes one after the other on the same core via the typical 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.

Based 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 either of them has to specify is the the difference between 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 more than the maximum size specified by the process. The starting address of this region is set as the contents of the base register. A address computed by the CPU is basically an offset added to the contents of the base register. The moment the CPU sends an address to the memory system, depending upon the process that is running, we generate addresses accordingly. Note that the contents of the base registers vary depending upon the process. In this system, if the process accesses an address that is beyond the limit register, then a fault is generated. A graphical description of the system is shown in Figure 2.8.

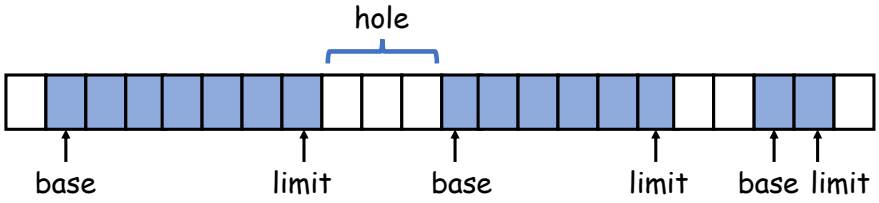


Figure 2.8: The base-limit scheme

We can clearly see 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 in 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.*

Definition 4 *Most memory management systems assign chunks of memory to processes. Sometimes some memory is wasted within a chunk. This wastage is referred to as internal fragmentation.*

Let us again take a deeper look at Figure 2.8. 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 what we need and then split it into an allocated region and a smaller hole. Very soon we will have a large number of these 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*. Of course, 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. Of course, the problem is that a lot of reads and writes will be 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 a 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 in which memory region 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 sent to the memory system. Even in this simple case, where we used a base and limit register, the address generated by the program is actually an offset that we need to add to the base register. This offset is actually the real address insofar as the CPU is concerned but this address is then modified (by adding it to the base) before sending it to the memory system. The gateway to the memory system is the instruction cache for instructions and the L1 data cache for data. This modified address is known as the *physical address*, whereas the address generated by the CPU is 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 5 *The virtual address is the address seen by the programmer, process, compiler and the CPU. In a k-bit architecture, it is also 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.*

A few ideas emerge from this discussion. Given a virtual address, there should be some sort of a table that we can lookup 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 one virtual address, or in other words there is a strict one-to-one mapping, then we 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 memory. 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 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.*

The 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 size and compatibility problems that we have seen earlier. It is possible to slightly repurpose the solution that we designed for the previous two problems to solve this problem as well. We will observe that repurposing our solutions is quite easy and in fact it is possible to run programs that have a larger memory footprint than the actual physical memory available on the system subject to some additional available space on storage devices such as the hard disk. The idea basically would be to use a part of the hard disk as an extension of the physical memory system.

It is time to summarize this subsection. We have basically 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 view of the real system. On a real system, we will see all of these problems because we have real-world constraints namely other component programs running and a small physical address space. We would thus like to formally introduce the term "Virtual Memory" here, which is simply defined as an abstraction of the memory space that the user perceives. The user in this case could be the programmer, compiler, running process or even the CPU. The virtual memory abstraction also incorporates a method to translate virtual addresses into actual physical addresses such that all three of our problems are solved.

Definition 6 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 that will from* 0 *to* $2^n - 1$*, where the memory address is assumed to be* n *bits. Practical implementations of the virtual memory abstraction solve the compatibility, overlap and size problems. The method for doing this is to somehow 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 6) 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 satisfy 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 Implementation of the 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* or a *cache line*. The memory system that comprises the caches and the main memory only deals with blocks. The advantage of this is that our memory system remains simple and we do not have to deal with a lot of entries in the caches. The reverse would have been true if we would have 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 and 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. Consequently, the mapping problem aims to map a page

Definition 7 *A page is typically a block of 4 KB of contiguous memory in the virtual memory space. Every page is 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 necessary 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 2.9. 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 in the physical address space.

The Page Table

Let us refer to the mapping table as the *page table*. Let us first look at a simple implementation of the page table, which is quite simple yet quite inefficient. Let us explain this 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. In a 32-bit memory system, we thus need 20 bits to specify a page address. We will thus have 2^{20} or roughly a million pages in the system. 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 lot and it represents a tremendous wastage of physical memory space. If we think about it, we shall observe that most of the virtual address

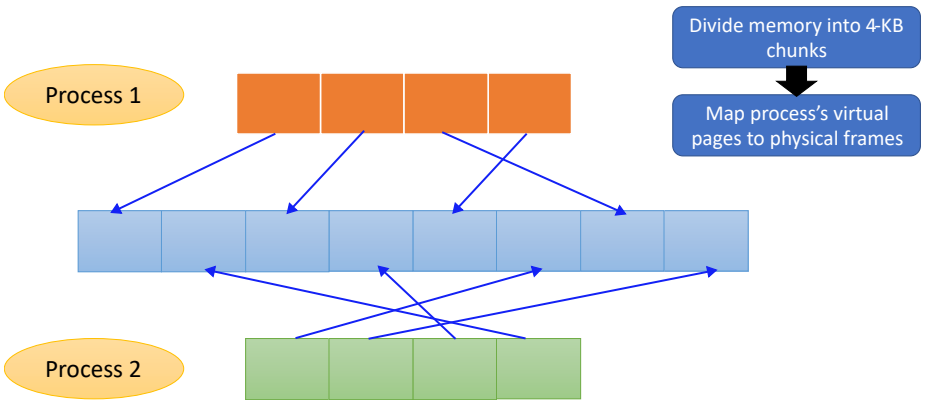


Figure 2.9: Conceptual overview of the virtual memory based page mapping system

space is actually not used. In fact, it is quite sparse particularly between the stack and the heap, which can actually be quite large. This problem is still manageable for 32-bit memory systems, especially if we don't have a lot of concurrently running processes. However, if we consider a 64-bit memory system, then the page table storage overhead is prohibitively large and clearly this idea will not work. Hence, we need far more efficient way of storing our mappings. We need to a look at the memory map of a process seriously and understand the structure of the sparsity to design a better page table (refer to Section 2.2.1).

Begin by noting that a very small part of the virtual address space is actually populated. The beginning of the virtual address is populated with the text, data, bss and heap sections. Then there is a massive gap. Finally, the stack is situated at the highest end of the allowed virtual memory addresses. There is nothing in between. Later on we will see that other memory regions such as memory mapped files can occupy a part of this region. But still we will have large gaps and thus there will be a significant amount of sparsity. This insight can be used to design a multilevel page table, which can capture leverage this pattern.

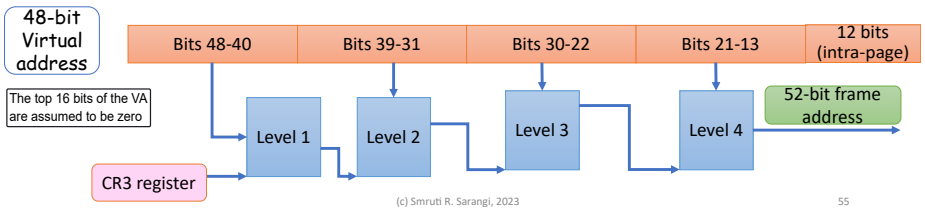


Figure 2.10: The design of the multi-level page table

The design of a multilevel page table is shown in Figure 2.10. It shows an example address translation system for a 64-bit machine. We typically observe that we don't need that large a virtual address. 2^{64} bytes is more than a billion gigabytes, and no practical system (as of today) will ever have so much of

memory. Hence, most practical systems as of 2023, use a 48-bit virtual address. That 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 48-bit physical 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. 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 2.11 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.

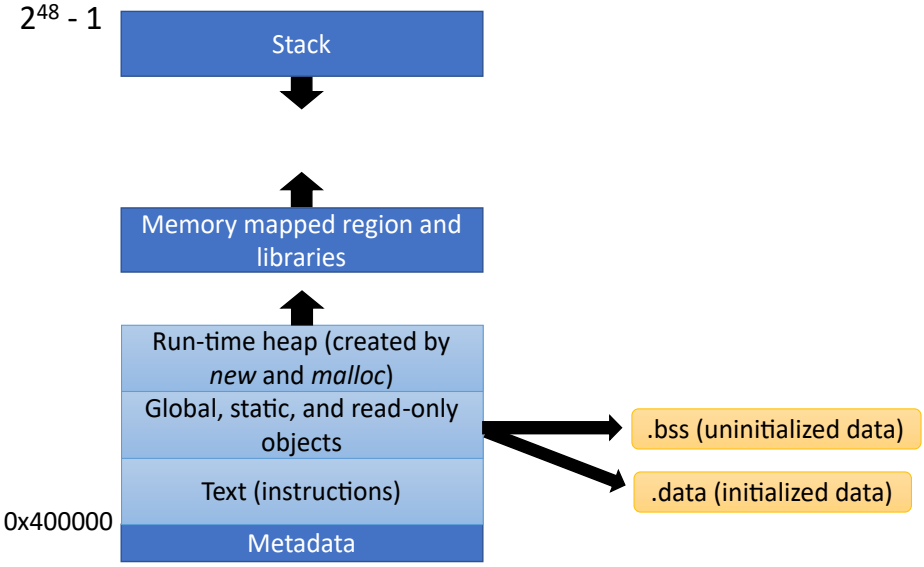


Figure 2.11: The memory map in 64-bit Linux

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 them 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 vary in their lower bits, however, in all likelihood their more significant bits will be the same. To cross-check in a decimal number system count from 0 to 999. How, frequently does the unit's digit change? 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 less significant bits.

Now, coming to the top level bits again (bits 40-48), we observe that 9 bits can be used to access 2^9 (=512) entries. Let us create a Level 1 page table that

is addressed by these 9 bits. An entry points to a Level 2 page table. However, given our explanation, we do not expect most of the entries in the Level 1 page table to point to a valid Level 2 page table. We instead expect that they will be empty. We need to store the address of the Level 1 page table itself to be stored somewhere. This is 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, needless to say, the first step is to read the CR3 register and find the base address of the Level 1 page table.

It is important to note that we only allocate that many Level 2 page tables as are required. If there are no addresses whose most significant 9 bits match a given sequence, then the corresponding entry in the Level 1 page table is empty. This saves a lot of space.

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 a 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 entries. We continue 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 36-bit frame address. This is the address translation.

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 is more than a 1000 cycles. This is a very slow operation. The sad part is that we need to do this for every memory access !!!

The TLB (Translation Lookaside Buffer)

Clearly, for every memory access, we cannot spend approximately 1000 cycles translating it. This is seems quite ludicrous. 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 memory translation latency. We need to understand that a page is actually a lot of memory. If we consider a 4 KB page, it can hold a 1024 integers. If we assume some degree of locality – temporal or 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. Note that there is much more locality at the page level than at the cache line level. As a result, we do not need to cache a lot of entries. In fact, it has been shown that caching 64 to 128 entries is good enough.

Hence, almost all processors have a small hardware cache called a TLB (Translation Lookaside Buffer). It caches 64 to 128 translation entries. It can also have two levels (L1 TLB and L2 TLB) and cache more entries (roughly a 1000). 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 in a modern core, this small latency is hardly perceptible and as a result, the

translation basically comes for free.

In the rare case, when there is a miss in the TLB, then 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. Note that the translated address in this case is a frame address, which can then be directly read from the memory system (caches and main memory).

Solution of 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 region of memory on a storage device such as the hard disk or a flash drive, or even the hard disk of a machine accessible over the network. This reserved region is known as the *swap space*.

Whenever we need more space than what physical memory can provide, we take up space in the swap space. Frames can be resident either on physical memory or in the swap space. However, for them to be usable, they need to be brought into main memory.

Let us now go over the 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 exists, then the physical address is sent to the memory system: instruction cache or L1 data cache. The access traverses the memory system until it reaches the main memory, which is guaranteed to have the data. However, in the rare case when an entry is not there in the TLB, we record a *TLB miss*. There is a need to access the page table, which is a slow process.

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

If a translation is not found in the page table, then we can have several situations. The first is that address is illegal. Then of course, an exception needs to be raised. However, it is possible that the entry indicates that the frame is not in memory. We can have a single bit for this, where 1 may indicate that the frame is in main memory and 0 may indicate that the frame is in the swap space of the hard disk. However, it is possible that we don't have a single swap space but a bunch of swap spaces or the swap space is on some other device such as the USB drive or on a machine that is accessible over the network. Hence, a page table entry can additionally store the location of the frame and the device that contains it. 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 is stored in the page table entry. However, it is not there in the TLB entry because in this case we assume that the frame is there in main memory. Let us look at some of the other fields that are there in a page table entry notably the permission bits.

Permission Bits in a Page Table Entry

The page tables and TLBs store more information as well. They store some permission information. For security reasons, a program is typically not allowed to write instruction-only or *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. It also makes it easier to find loopholes. We can thus associate three 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 on 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-mode pages. Given that the mapping information will never come from the page table to the TLB, the CPU running a user mode program will never be able to access a kernel-only page.

2.2.4 Segmented Memory

Between the purely virtual and physical abstractions of memory, we can add more layer called segmentation. The x86 architecture has a set of segment registers that add a layer of abstraction between the virtual and physical layers. Consider the case of instructions. The address of an instruction issued by the CPU is a virtual address. This address is added to the contents of the *code segment* register. In this case, we are effectively adding an offset to the virtual address. Let us refer to this as a *segmented address*. This segmented address is then translated to a physical address using the regular address translation mechanism. We thus have inserted an extra layer between purely virtual and physical memory.

On similar lines, we have a data segment and stack segment register. For example, any data access is similarly an offset from the address specified in the data segment register. The same holds for an address on the stack.

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 for instance prohibit any data access from accessing code page. This is easy to do. We have seen that in the memory map the lower addresses are code addresses and once the 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. Segmentation is an easy way of preventing such attacks. In many other attacks, it is assumed that the addresses of variables in the virtual address space are known. For example, the set of attacks that try to modify return addresses stored on the stack 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 in a given address on the stack – it will change in every run. Note that program correctness will not be hampered because a program is compiled in a manner where it is assumed that all addresses will be computed as offsets from the contents of their respective segment registers.

There are some other ingenious uses as well. It is possible to define small memory regions that are *private to each core (per-core regions)*. The idea here is that it is often necessary to store some information in a memory region that is private to each core and should not be accessed by processes running on other cores, notably kernel processes. These regions are accessed by kernel threads mainly for the purposes of memory management and scheduling. An efficient way of implementing this is by associating an unused segment register with such a region. All accesses to this region can then use this segment register as the base address. The addresses (read offsets) can be simple and intuitive such as $0, 4, 8, 12$, etc. In practice, these offsets will be added to the contents of the segment register and the result will be a full 64-bit virtual address.

Segmented Addressing in x86

Figure 2.12 shows the segment registers in the x86 ISA. 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 can for instance be used to store per-core data that is not meant to be accessed via programs running on other cores.

Figure 2.13 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 the ids of the segment registers for the current process. Most of the time, the values of the segment registers will be found in the SDC. Segment register values get updated far more infrequently as compared to TLB values. Hence, we expect hits here almost all the

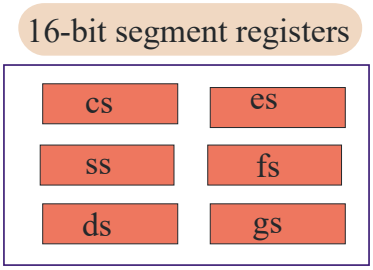


Figure 2.12: The segment registers in the x86 ISA

time other than some rare instances when a process is loaded either for the first time or after a context switch.

Similar to a TLB miss, if there is a miss in the SDC, then there is a need to search in a larger structure for a given segment register belonging to a process. In older days there used to be an LDT (local descriptor table) and a global descriptor table (GDT). We could think of the LDT as the L1 level and the GDT as the L2 level. However, nowadays the LDT is mostly not used. 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. Of course, it also has a finite capacity, and if there is a miss there then an interrupt is raised. The operating system needs to populate the GDT with the correct value. It maintains all the segment register related information in a dedicated data structure.

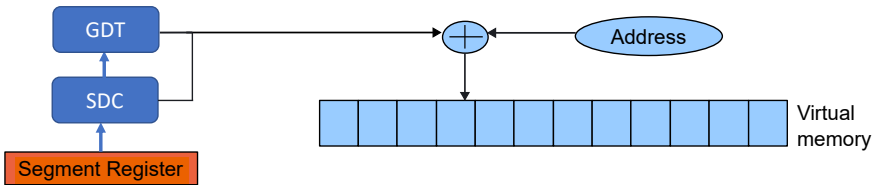


Figure 2.13: Computing the virtual address with memory segmentation

As seen in Figure 2.13, the base address stored in the relevant segment register is added to the virtual address. This address further undergoes translation to a physical address before it can be sent to the physical memory system.

2.3 I/O System

Any computing machine needs to read inputs from the user and needs to show the output back to the user. In other words, there needs to be a method to interact with it. 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 or there could even be bidirectional transfer such as a network device. It should be easy for a 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 comes with some performance guarantees. 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 interface chips and ports.

2.3.1 Overview

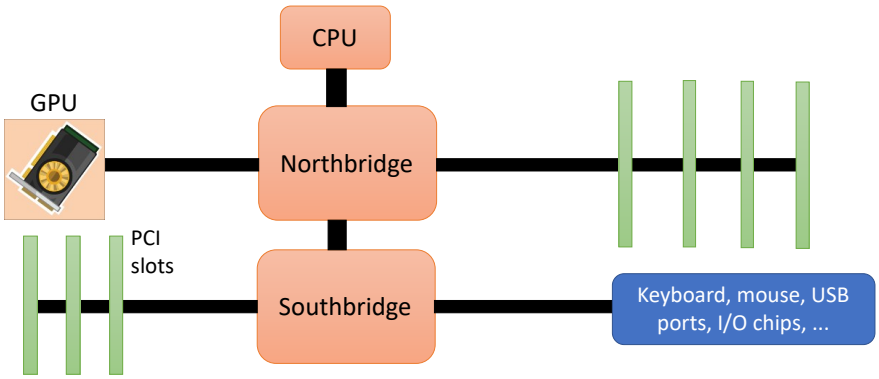


Figure 2.14: Overview of the I/O system

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. The reason that we need so many pins is because modern processors draw a lot of current. A pin has a limited current delivery capacity. However, a few hundred bits 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 2.14 shows a traditional design where the dedicated circuitry for communicating with the main memory modules and the GPU are combined together into a 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 chip has moved into the 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 and hard disk a dedicated controller chip called the Southbridge chip is used. In most modern designs, this chip is outside the chip – it is placed on the motherboard. The simplest way is to have a Northbridge-Southbridge connection. However, this is not mandatory. There could be a separate connection to the Southbridge chip and in high-performance implementation, we can have the Southbridge logic inside the CPU chip. Let us however stick to the simplistic design shown in Figure 2.14.

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 the messages to the PCI-X chip, which subsequently send the messages down the PCI-X buses to the target I/O device. The Southbridge chip may also choose to send a message to the USB ports, and a dedicated controller may then route the message to the specific USB port that the message is meant to be sent to.

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 tradeoff between the ease of programming, latency and bandwidth.

2.3.2 I/O via I/O Ports

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. All the devices and the controllers that are connected to the system expose themselves as I/O ports to software (read executables and assembly programs). The controllers in the chipset know how to route messages between the CPU and I/O ports. There are typically 2^{16} (65536) I/O ports; the mapping between a device and I/O port is set during boot time. 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. It queries the chipset for all the connected I/O devices and records their I/O ports. This information is made available to device drivers – specialized programs that within the OS that communicate with I/O devices.

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 2 consecutive I/O ports and a 4-byte access reads/writes 4 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 the 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, 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.

2.3.3 Memory-Mapped I/O

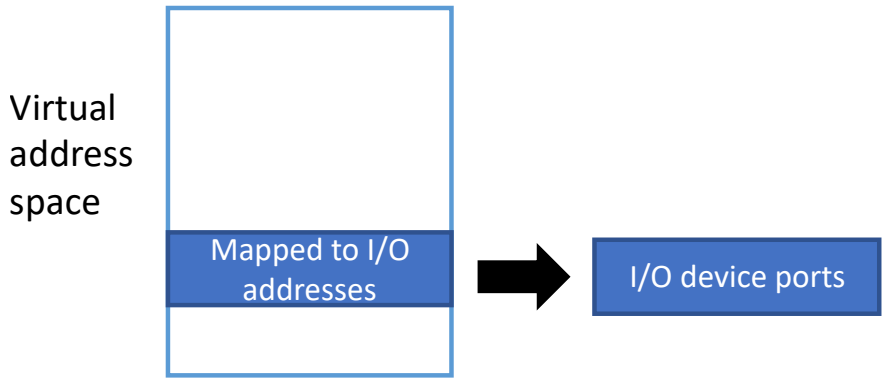


Figure 2.15: Memory-mapped I/O

The faster method is to map 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 2.15).

The first is that we can send a large amount of data in one go. The x86 architecture has instructions that allow the programmer to move hundreds of bytes between addresses in one go. These instructions can be used to transfer hundreds of bytes or a few kilobytes to/from I/O space. The hardware can then use fast mechanisms to ensure that this happens as soon as possible. This would mean reading or writing a large amount of data from memory and communicating with I/O devices.

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. 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 other chips in the chipset. The data that is 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 2023. In many reference manuals, it is conveniently referred to it via its acronym MMIO.

2.3.4 I/O using DMA

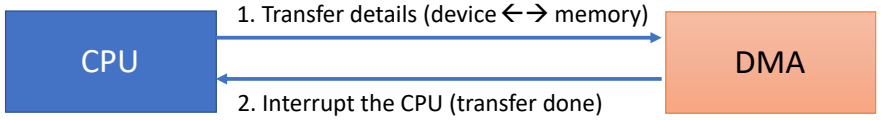


Figure 2.16: I/O using DMA

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 will fill up and the processor will remain stalled until the outstanding I/O operations complete. Of course, one simple solution is that we do the memory mapped I/O operations in smaller chunks; however, some part of this problem will still remain. 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 2.16, where there is a separate DMA (direct memory access) chip that effects the transfers between the I/O device and memory. The CPU basically outsources the I/O operation to the DMA chip. The chip is provided 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 arrives, the OS knows what to do with it and how to signal the device drivers that the I/O operation is done and they can 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".

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. Nevertheless, I/O ports are still used mainly 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 on the strength of the virtual memory system to provide a very convenient and elegant interface for device drivers. 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 and it is important to note that the key strength of memory-mapped I/O is that it is not required.

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 simply 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 has been completed.

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 (stored on a storage device like the hard disk), 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, and then 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 continues 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 in many senses has a life of its own from creation to destruction with numerous context switches in the middle.

A process during its execution can acquire a lot of resources; these are released once the process terminates. Some of the 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 a certain service from the OS, whereas signals are used by the OS to communicate information to a process.

This chapter has four sub-parts. 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.

In the second part, we shall further delve into this topic and specifically focus on the kernel's process descriptor (known as a task descriptor). We shall discuss the relevant code for managing the process ids (*pid*s in Linux), the process state and 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 Emde Boas tree.

In the next sub-part, we will look at the methods of process creation and

destruction. Specifically, we will 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 new process on top of the currently running process. This is the standard mechanism by which new processes are created in Linux.

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

3.1 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, the simple description does not yield to a simple implementation. We need to create elaborate data structures to store fields associated with the process especially the resources that it owns throughout the system and their usage status.

There can be many types of processes. To start with, we can have a single-threaded process or a multi-threaded process. A *thread* is defined as a *lightweight process*, which basically means that 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. Basically a thread is a process in its own right but it can also be a part of a thread group (multi-threaded process), where the different threads share a large part of the memory space and system resources. The only part of the memory space that is not shared is the stack – every thread has a private stack. Most operating systems including Linux view a thread as a process that executes on its own. It is just that it is a special kind of process where it is interrelated with other threads and all these threads form a thread group, which can also be viewed as a large multi-threaded process (with a single thread group id or `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.

3.2 The Process Descriptor

3.2.1 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`.

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`

3.2.2 `struct thread_info`

Overview of Low-Level Data Types and Structures

`thread_info` 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 very essential to align the fields of the data structure with actual memory words such that they can be accessed very efficiently and even their physical location in memory can be leveraged. 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 it. The Linux kernel is designed to run on a large number of machines that have very different instruction set architectures – in fact some maybe 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 bytes-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 that for instance 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 ŭ64 that correspond to unsigned 32-bit and 64-bit integers, respectively, on all target architectures. It is the job of the architecture-specific programmers 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 are required to implement other parts of the booting 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.

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. 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

```
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 sub-optimal 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. However, given that we allow multiple threads or a thread group to also be referred to as a 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.

3.2.3 Task States

Let us now look at the process states in Linux. This is shown in Figure 3.1. In the scheduling world, it is common to refer to a single-threaded process or a thread in a multi-threaded 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 look at the states that a task can be in.

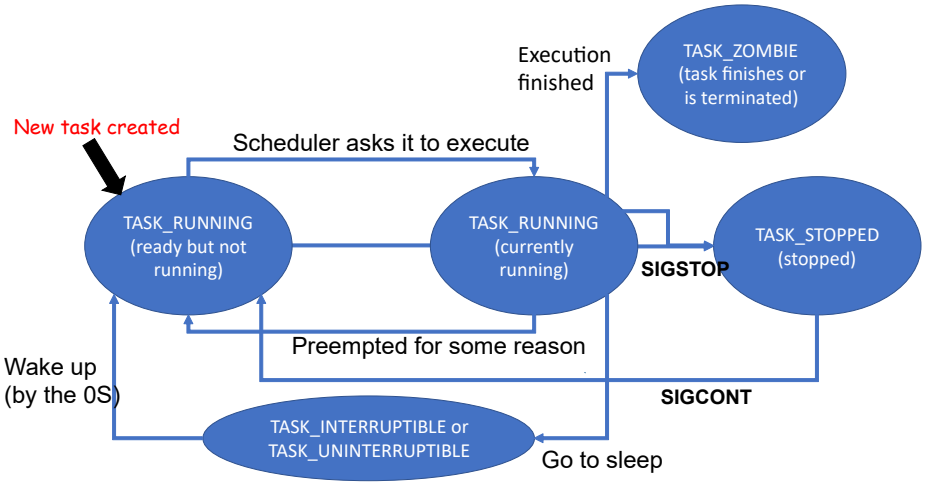


Figure 3.1: Linux task states

Here is the fun part in Linux. A task that is currently running and the task that is running 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. We shall see later that this simplifies many design decisions. Furthermore, 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 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 using the `SIGSTOP` signal. Specifically, the `kill` system call 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 doing this, 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 also be used to resume such a suspended task.

Let us now come to the two interrupted states: INTERRUPTIBLE and UNINTERRUPTIBLE. A task enters these states when it requests for some service like reading 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 wrapping up a task is quite elaborate in Linux. Recall that the processor has no way of knowing when a task has completed. It is thus necessary to explicitly inform it 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. 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.

3.2.4 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. It has a dedicated kernel stack that the kernel thread uses when it begins 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.

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

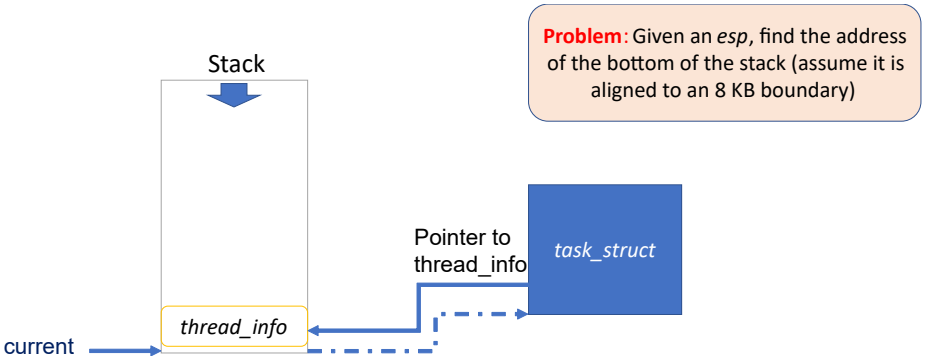


Figure 3.2: The structure of a kernel stack (older versions of Linux)

For a long time, the kernel stack was limited to two pages, i.e., 8 KB. It contained useful data about the running thread. These are basically per-thread stacks. In addition, the kernel maintains a few other stacks, which are CPU specific. 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. There is thus a need to switch to a new interrupt stack. Hence, each CPU has an interrupt stack table with seven entries. This means that Linux can handle deeply nested interrupts (till 7 levels). This is conceptually similar to the regular context switch process for user-level tasks.

Figure 3.2 shows the structure of the kernel stack in order kernels. The `thread_info` structure was kept at the lowest address and there was a dedicated `current` pointer that pointed to the `thread_info` structure. This is a very quick method of retrieving the `thread_info` associated with the current task. In fact from any stack address, we can quickly compute the address stored in the `current` pointer using the fact that the starting address of `thread_info` needs to be a multiple of 8 KB. Simple bitwise operations on the address can be used to find this value (left as an exercise for the reader). Once, we get the address of the `thread_info` structure, we can get the pointer to the `task_struct`.

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 of 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 segmentation. 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 has the potential for being frequently used. Furthermore, to use segmentation, 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

```
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 (works like a macro from a performance point of view). The most important thing is that this pointer to the current *task_struct*, which is accessible using the `current` variable, is actually stored in the `gs` segment [Lameter and Kumar, 2014]. This serves as a dedicated region, which is specific to a CPU. We can treat it as a per-CPU storage area that stores a lot of information that is relevant to that particular CPU only.

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 of the segmentation mechanism in x86. 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 (may not be portable), or even a known memory address. The issue with storing this pointer at a known memory address is that it significantly limits our flexibility in using the virtual address space and there are 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. No cache line stores data corresponding to a per-CPU region and other data. If this is the case, we will have a lot of *false sharing* misses across the CPU cores, which will prove to be very 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.

3.2.5 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. This is because 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.

3.2.6 Computing Actual Task Priorities

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

```
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 represents 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 done in the code snippet that is 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 removed. We need to ensure monotonicity.

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 in a similar manner. 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` effects the 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. The superuser is a special kind of a user who has more privileges and is supposed to play the role of a system administrator. She still does not have kernel privileges but she can specify a negative nice value that is between -1 and -20. However, she still cannot raise the priority of a regular user-level process to that of a real-time process, but she can arbitrarily alter the priority of all user-level processes. 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 the actual priority. This number is henceforth used throughout the kernel code to represent actual task priorities. We will see that when we discuss schedulers, the process priorities will be very important and shall play a vital role in making scheduling decisions.

3.2.7 `sched_info`

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

```
/* # 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 run queue. Note that the *run queue* is a structure that stores all the tasks whose status is TASK_RUNNING. 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` that store when a task last ran on the CPU and it was last queued to run. In general, the unit of time within a CPU is either in milliseconds or in jiffies (refer to Section 2.1.4).

3.2.8 Memory Management

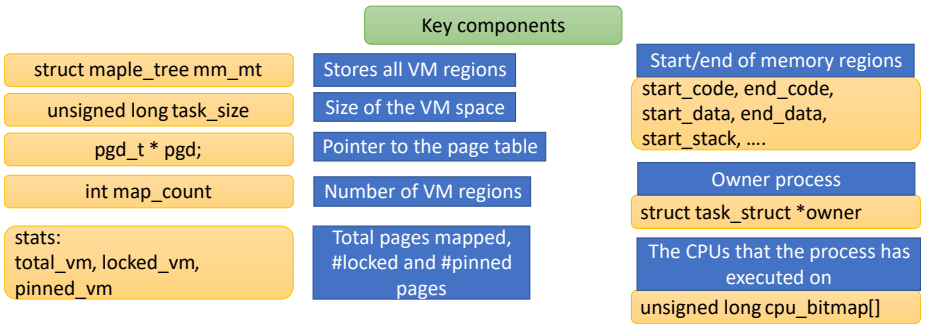


Figure 3.3: The key components of a process's address space

A process typically owns a lot of memory space. It owns a bunch of virtual memory regions and a page table that is heavily annotated with a lot of information. This can be security information but also can be performance or correctness-related hints such as information related to locking and pinning (see [Corbet, 2014]).

For instance, we may want to *lock* a set of pages in memory (make it very hard to swap them to disk). This will eliminate page faults for those set of pages. On the other hand, we can also *pin* the pages in memory, which is slightly more restrictive. It does not allow the kernel to move the page 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. For example, we would want the pages of the page table to be *pinned*. This way, we would exactly know where they are and this information will remain stable throughout the execution of the process. The kernel can access the page table using its physical address. 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 the memory-related information as shown in Figure 3.3. `mm_struct` additionally stores the locations of all the sections of the process in memory. Recall that we had discussed the way in which an object file and a binary are divided into sections in AppendixB. This information needs to be stored in a dedicated data structure that is accessible to the kernel. Next, we need to store

the id of the owner process (pointer to a `task_struct`).

The last field `last_queued` 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 the 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 the 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.

3.2.9 Storing Virtual Memory Regions

It is important to keep track of all the virtual memory regions that a process *uses*. 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 very large sparse areas small chunks of virtual memory are used.

Many of these 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 of the used virtual memory regions. Given a virtual memory address, it should be possible to very easily 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

```
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 start and end addresses. It also contains a pointer to the original `mm_struct`. Let us now 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 for instance.

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 do not have a file-backed copy 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. 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 mesh of linked lists. We will study this later when we discuss physical memory allocation in detail. Till now it suffices to say that there is a linked list pointed to by `anon_vma_chain` to store these regions. Basically, there is a pointer from `vm_area_struct` to the corresponding region in the linked list of anonymous regions.

We will now request the reader to kindly take a look at some of the important data structures that are used in operating systems (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 in memory systems.

Specifically, the reason that we need all of these tree-based data structures is as follows. Many time, we face problems like identifying the VM area structure that contains a given virtual memory address. This boils down to search problem. Often using a hash table is not a very wise idea, particularly when we are not sure of how many keys we need to store. Trees work better because of performance reasons. Trees are very scalable and provide good performance as long as they remain balanced. We can always use the classical red-black and AVL trees. However, it is far more common to use B-tree based structures especially for storing a lot of data. These are balanced trees and provide their results in logarithmic time.

The Radix tree is also a very useful data structure, which traverses the tree based on the digits/letters in the key. The search time is independent of the number of nodes. Another useful structure is the Van Emde Boas tree that helps us solve problems of the following type: given a bit vector, find the location of the first 0 or 1 in logarithmic time. It is an efficient search structure that is superposed on a large bit vector.

3.2.10 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 the 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.

Managing all the *pid*s is an important problem. Whenever a new process is started, we need to allocate a new *pid* and whenever a process is torn down, 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. It defaults to 32,768.

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

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?

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`, we use 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 tried a large number of data structures. They found that most of the time, processes share prefixes in terms of their more significant digits. This is because, most of the time, the processes that are active roughly have a similar *pid* range. As a result, if let's say we have looked up one process's entry, the relevant part of the data structure is still present in the processor's caches. It can be used to quickly realize a lookup, given that the subsequent *pid*s are expected to share prefix digits. Hence, in practice, such Radix trees were found to be faster than hash tables.

3.2.11 Processes and Namespaces

Linux groups processes into *namespaces*. A namespace is basically a set of processes where the processes can only see each other and they cannot see processes in other namespaces. We can think of a namespace more as an *isolated* group of processes. There are many reasons for creating a namespace.

Many of these stem from some modern computing paradigms such as microservices, containers and serverless computing. We wish to create a small isolated environment within a machine that runs a traditional operating system. This is done mostly for security reasons, where we want to create a barrier between a process and the rest of the system. We need to note that the goal here is to securely execute a process on a remote system because the code is not completely trustable. Thus there is a need to 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. Hence, the need for technologies such as namespaces arose.

Almost all modern versions of Linux support the notion of *containers*. A container is a set of processes along with a set of resources such as a file system and network connections. A namespace is typically associated with a container and encompasses all the processes that are running within it. Now bear in

mind that a container at any point of time can be stopped, suspended and migrated to another machine. On the other machine, the container can be restarted. We want processes to continue to have the same process ids even on the new machine. This will ensure that processes execute unbeknownst to the fact that they have actually been migrated. Furthermore, if they would like to communicate with each other, it is much more convenient if they retain the same *pid*s. This is where the notion of a namespace comes in handy.

In this case, a *pid* is defined only within the context of a namespace. Hence, when we migrate a namespace along with its container, and then the container is restarted on a remote machine, it is tantamount to reinstating the namespace and all the processes within it. Given that we wish to operate in a closed sandbox, they can continue to use the same *pid*s and the operating system on the target machine will respect them because they are running within their separate namespace.

A namespace itself can be embedded in a hierarchy of namespaces. This is done for the ease of management. 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 them 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

```
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 a little bit of 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 large number 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 an object. 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

```
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 possibly 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.

A `struct pid` can also be used to identify a group of tasks. This is why a linked list of the type `tasks` is maintained. The last field `numbers` is very interesting. It is an array of `struct upid` data structures (also 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 namespaces, the same process (identified with `struct pid`) can have a different process id number (*pid* value). Hence, there is a need to store such ⟨*pid*,namespace⟩ tuples.

Allocating a `pid` Structure

Let us now look at the process of allocating a new process id. We start out with invoking the `alloc_pid` function defined in kernel/pid.c. It starts out with trying to get a free `struct pid` structure from the software pool. This is always a method of choice because it is very fast and a pool also helps detect memory leaks.

The next step is to allocate a *pid* number in each namespace that the process is a part of, which includes its default namespace as well as all ancestral namespaces. Basically, the philosophy 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 ancestor namespaces, access their `idr` trees and creative mapping between the newly allocated `pid` structure and a *pid* number. Note that a *pid* number is only defined within its namespace, not in any other namespace. The *pid* number should ideally be the lowest unallocated *pid* number within that namespace.

The `idr` tree is actually more than a Radix tree. Its nodes are annotated with more information. 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 achieved by the Radix tree. However, the `idr` tree does more in terms of finding unallocated *pid*s as well. It is also a van Emde Boas tree (see Appendix C), which helps find the the lowest unallocated *pid* number in logarithmic time. This functionality is achieved as follows.

First, to index the Radix tree part of the `idr` tree, we start from the more significant MSB bits of the *pid* and gradually proceed towards the LSB bit. This ensures that the leaves of the tree that correspond to distinct mapped *pid*s are in sorted order if we traverse the tree using a preorder traversal. Each leaf thus corresponds to a mapped *pid*. We then annotate each leaf with a pointer to the corresponding `struct pid`. This helps us realize a key-value table, where the key is the *pid* number and the value is the `struct pid`.

Let us now appreciate the fact that we can actually do more. We start with maintaining a bit vector corresponding to every process in the namespace. Each leaf or even an internal node is uniquely identified by the path that we follow from the root to reach it. This is basically the prefix of his binary representation (starting from the MSB). For a leaf node, this prefix is equal to its *pid* number, which is the defining property of a Radix tree. Now, given the prefix, we can implicitly map it to a point in the bit vector. For *pid* p we map it to the p^{th} entry of the bit vector.

Let us explain with an example shown in Figure 3.4. There are four *pid*s: 1, 3, 4 and 7. Their binary representations are 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 3.4.

The internal nodes are shown as ovals and the leaf nodes are rectangles. Each leaf node is uniquely identified by the path from the root leading to it (the *pid* number). Each leaf node additionally points to the `struct pid` associated with the process. Along with that each leaf node implicitly points to a location in the bit map. If a leaf node corresponds to *pid* 3 (011), then we can say that it maps to the 3^{rd} bit in the bit vector.

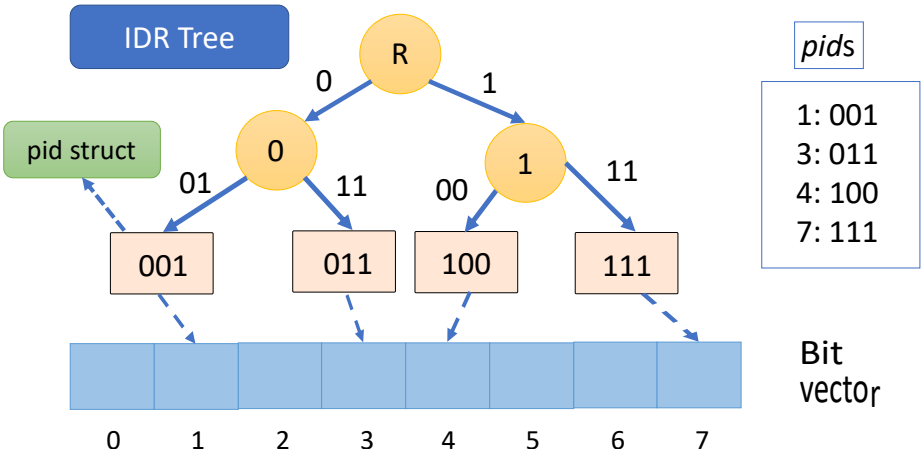


Figure 3.4: Example of an IDR tree

Let us now interpret the IDR tree differently. Each leaf node corresponds to a range of *pid* numbers. For instance, we can associate *pid* 1 with the indices 0 and 1 in the bit vector. Similarly, for *pid* 3, we can associate all the indices after the previous *pid* 1. These will be indices 2 and 3. Similarly, for *pid* 4, we can associate index 4 and for *pid* 7, we can associate 5, 6 and 7.

This is just an example, we can associate larger indices until the next *pid* (in the ascending/preorder sequence of leaves). The convention that is used does not matter. We are basically creating non-overlapping partitions of the bit vector that are in sorted order.

We interpret the bit vector as follows. If a bit is equal to FREE (logical 1 in the case of x86), then it means that the corresponding *pid* number is free, and vice versa. Each internal node stores similar information as a van Emde Boas tree (vEB tree). The aim is to find the index of the leftmost (lowest) entry in the bit vector that stores FREE. Note that in this case a node can have more than 2 children. For each child we store a bit indicating if it has a free entry in the subtree rooted at it or not. Let us say that there are k children. The parent stores a k-bit vector, where we start searching it from index 0 to $k-1$. We find the earliest entry whose status is FREE. The algorithm recursively proceeds using the subtree rooted at the chosen child. At the end, the algorithm reaches the leaf node.

Each leaf node corresponds to a contiguous region of bits in the bit vector.

This region can be very large especially if a lot of contiguous *pid*s are deallocated. Now, we need to note that there is no additional vEB tree that additionally indexes this region. Other than the Radix tree nodes, we don't have additional nodes. Hence, we need a fast method to find the first location that contains a FREE entry in a large contiguous chunk of bits in the bit vector.

Clearly, using a naive method that scans every bit sequentially will take a lot of time. However, some smart solutions are possible here. We can start with dividing the set of contiguous bits into chunks of 32 or 64 bits (depending on the architecture). Let us assume that each chunk is 64 bits. We can typecast this chunk to an unsigned long integer and compare it with 0. If the comparison succeeds, then it means that all the bits are 0 and there is no 1 (FREE). If it is non-zero, then it means that the 64-bit chunk has at least one 1. Fortunately, x86 machines have 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 1-2 cycles. The kernel uses this instruction to almost instantaneously find the location of the first 1 bit (FREE bit).

As soon as a FREE bit is found in the bit vector, 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 vEB 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 entry from 0 to 1 (FREE) and appropriate changes are made to the nodes in the path from the leaf to the root.

3.2.12 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 filesystems. They organize their constituent files and directories differently based on considerations such as read-write patterns, sequential vs random access, 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 request that have been sent to different block-level 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 regarding block devices. bio stands for
       block I/O*/
    struct bio_list *bio_list;

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

3.2.13 The PTrace Mechanism

There is often a need for a parent process to observe and control the execution of a child process. This is often 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 all right. 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. The reason we are so concerned about system calls is because this is the primary mechanism by which a process interacts with the operating system. If its intent is malicious, then that 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. 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, many a time 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 is 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.

3.3 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 weird

to beginners, 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 (notably) has other mechanisms.

This model is simple and to many programmers it is much more intuitive. The kernel defines a few special processes notably the *idle process* that does nothing and the *init* process that is the mother process for all the processes in the system. The idle process is basically more of a concept that an actual process. It represents the fact that nothing active is being done or in other words no program is running. Its *pid* is 0. The *init* process on the other hand is the first process to run (started at boot time). It spawns the rest of the processes. Its *pid* is 1.

3.3.1 The Fork Mechanism

The *fork* system call is arguably the most famous in this space. The *fork* call creates a clone of a running process at the point that it is called. It is basically a straightforward copy. 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. Pretty much the parent creates a twin of itself. However, after returning from the *fork* call, the child and parent go their separate ways. For all practical purposes, they are separate processes and are free to choose their execution paths.

Before we delve more into the *fork* system call, let us add a word about the *init* process. The key boot time process is called `start_kernel`. Its job is to initialize the kernel as well as all the connected devices. It basically does the role of booting. It is defined in init/main.c. Let us 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. It is thus born as a kernel-mode process. However, it quickly transitions to the user mode and remains a user-mode process throughout. 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 is created and all the necessary user-level processes have been created, the boot process ends. After this, there are a fair number of user-level processes. We can then interact with the system and launch applications. Note that are applications are again created using exactly the same mechanism. Some process that has already been created is forked to create our application process. After creation, we are not bound to use the variables defined by the parent process or execute the code of the parent process. A child process becomes independent and it is free to execute any program, 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 (`pid` in the code) after creating the child process.

Note 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 is fully copied to the child. For example, if a variable `x` is defined to be 7 in the code prior to executing the *fork* call, then after the call is over and the child is created, both of them 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 – it has different physical addresses even though its starting value is the same (i.e., 7). This means that if the parent changes `x` to 19, the child will still read it to be 7 (because it possess a copy of `x` and not `x` itself). Basically, the child gets a copy of the value of `x`, not a reference to it. 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. A natural question that can be asked here is that prior to executing the *fork* call, there was no child, how can it 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 full task 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 the variable `pid` and start executing Line 8 in Listing 3.9. The notion of forking a new process is shown in Figure 3.5.

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

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

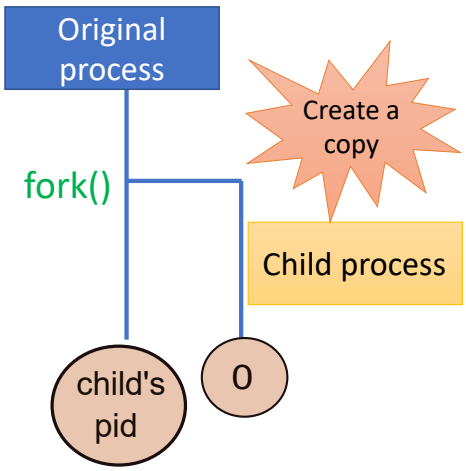


Figure 3.5: Forking a child process

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. After this happens, 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 in 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

Figure 3.6(a) shows the copy-on-write mechanism. In this case, we simply copy the page tables. The child inherits a verbatim copy of the parent's page table even though it has a different virtual address space. This mechanism ensures that the same virtual address in both the child and parent's virtual address space points to the same physical address. This basically means that no memory is wasted in the copying process and the size of the memory footprint remains exactly the same. Note that copying the page table implies copying the entire memory space including the text, data, bss, stack and heap. Other than the return value of the *fork* call, nothing else differentiates the child and parent. They can seamlessly read any address in their virtual address space and they will get the same value. However, note that this is an implementation hack. Conceptually, we do not have any shared variables. 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. It is true that to save space as well as for performance reasons, the same piece of physical memory real estate is being used, however conceptually, these are different variables. This becomes very clear when we consider write operations.

This part is shown in Figure 3.6(b) where we see that upon a write, a new

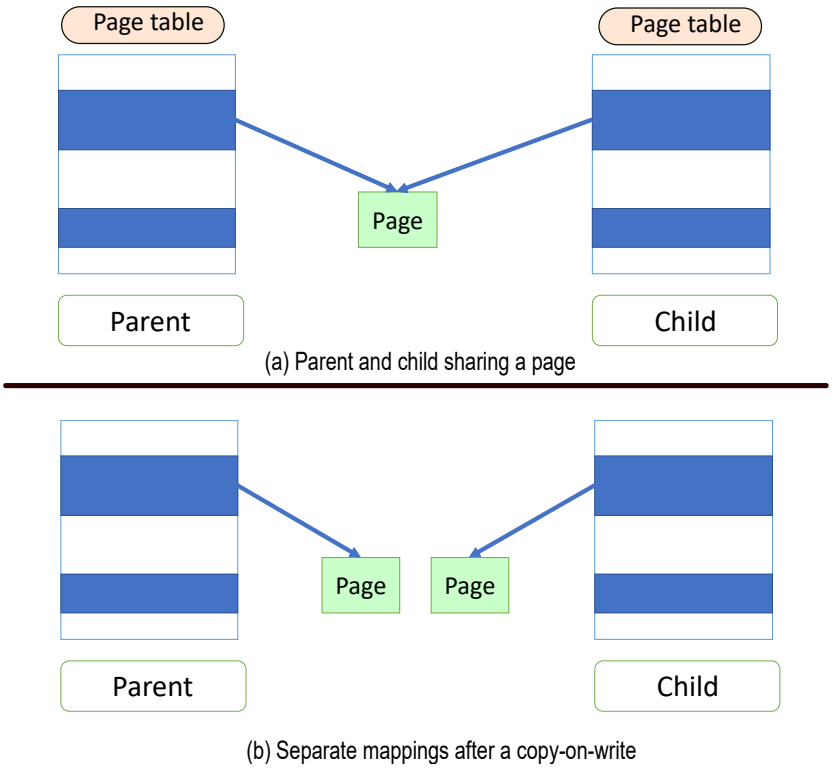


Figure 3.6: The copy-on-write mechanism

physical copy of the frame that contains that variable that was written to is created. The child and parent now have different mappings in their page tables. Their page tables are updated to reflect this change. For the same virtual address, they now point to different physical addresses. Assume that the child initiated the write, then after a new physical copy of the page is created and the respective mapping is made, the write is realized. In this case, the child writes to its 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 no write. This saves space and also allows us to fork a new process very quickly. However, the moment there is a write, there is a need to create a new physical copy of the page and realize the write on the respective copy (copy on write). This does increase 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 often not visible.

There are several reasons for this. The first is that the parent or child may not subsequently write to a large part of the memory space. In that case, the copy-on-write mechanism will never kick in. The child may overwrite its memory image with that of another binary and continue to execute it. In this case also, there will be no need for a copy-on-write. Furthermore, lazily creating copies of pages as and when there is a demand, distributes the overhead over a a long

period of time. Most applications can absorb this 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? This is where we use the permission bits in virtual memory. 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 OS can catch and take appropriate action. It needs to perform a copy-on-write and reset the read-only status for both the parent's and child's version of the page. Let us now delve into the fine print.

It is possible that the process already has some pages such as code pages, which are set to be read-only. Their `READONLY` bit is set. This information needs to be preserved, otherwise we may erroneously reset the read-only status of such pages, which should never be done. 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 TLB and 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 is invoked and a new copy of the page is created. Now, we can set the `P2` bit of the page in both the parent and child's page table to 0. The need to track write accesses to it is not there anymore. A separate copy has already been created and the parent and child can happily read/write to their respective private copies.

Details

Let us now delve into some of the details of the *forking* mechanism. We would like to draw your 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. 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 multi-threaded process (thread group), only the calling thread is forked.

Inside the kernel, all of them ultimately end up calling the `copy_process` function in kernel/fork.c. The signature of the 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 the new 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) Assigned 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* in its namespace
2. Copy of all the information about open files, network connections, I/O, and other resources from the original 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 virtual address map and other memory related information (`struct mm_struct`). This also copies the page table.
   (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 in the child task
   (a) Add the new task to the list of children in the parent task
   (b) Fix the parent and sibling list of the newly added child task
   (c) Add the thread group information in a task that belongs to a multi-threaded process.

3.3.2 The exec Family of System Calls

It is important to note that after returning from a *fork* operation, the child and parent process can go their separate ways. For example, the child may decide to completely replace its execution state and start executing a new binary 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 on a terminal. It accepts our user inputs and starts the process of execution of a binary executable. 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

```
#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 by default is 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. There are a bunch of 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 (process 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, data and bss sections. Then there is a need to initialize the stack and heap sections as well as initializing the starting value of the stack pointer. In general, connections to external resources such as files and network addresses are preserved in an *exec* call. Hence, there is no need to modify, cleanup or reinitialize them. We can now start executing the process with an updated memory map from the start of the text section. This is like regular program execution. The fact that we are starting from a forked process is conveniently forgotten.

3.3.3 Kernel Threads

Linux distinguishes between user threads, I/O threads, and kernel threads. We have already looked at user-level threads. I/O threads are reasonably low-priority threads, whereas 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 (represented by its `task_struct`) is a kernel thread. Linux defines analogous functions like `kthread_create` and `kernel_clone` to create and clone kernel threads, respectively.

Similar to *init* in the userspace that acts like a mother process, *kthreadd* acts like a mother process for all kernel threads. Its *pid* number is 2. Kernel threads do the job of the operating system. They are primarily used for implementing all kinds of bookkeeping tasks, timers, interrupt handlers and device drivers.

3.4 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 that the scheduler chose is a very involved process. We can end up resuming the execution of the same process that was paused or it could be another process. In either case, the algorithm is the same.

3.4.1 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 this needs to be correctly save understood 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 and there is no place for even going wrong with a single bit. In the context of x86-64, let us delve into the meaning of the *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 and return address register (if the architecture has one)
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 component of the hardware context in a large and complex processor like an x86-64 machine. However, we have listed the main components for the sake of readability. This *context* needs to be stored and later restored.

It is important to discuss the two virtual memory related structures here namely the TLB and page table. The TLB 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. Instead, every process is made to use only its own mappings. Because of the *pid* information that is present, a process cannot access the mappings of any other process. This mechanism

reduces the number of TLB misses; as a result, there is a net performance improvement.

The page table, open file and network connections and the details of other resources that a process uses are a part of the *software context*. They remain in the process's `task_struct`. There is no need to manage them upon a context switch because they do not need to be specifically stored and restored later.

3.4.2 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, there are four types: kernel → kernel, kernel → user, user → kernel and user → user. For the sake of discussion, let us consider the case of a user-level process because it is more common. As we have discussed earlier, a context switch can be triggered by three events namely an interrupt, an exception or a system call. Furthermore, we have a method of creating 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 to run and do its job. One of the primary tasks of the kernel henceforth 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 a context switch causing event 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 coming in 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. In fact, we can reuse the same user-level thread that was just suspended. Specifically, the same `task_struct` can be used and the thread can simply be run in the "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, its stack changes and becomes a kernel stack. The user-level stack is not used in kernel

mode. These are some of the minimal changes that need to be made. This method is clearly performance-enhancing and in character it is very lightweight.

Also, there is no need to flush the TLB or change the page table. If we agree to use a different set of virtual addresses in kernel mode, then there is no problem. This is because the kernel code that will be executed will only use virtual addresses that are in the kernel space and have no overlap with user-level virtual addresses. This kernel code is loaded separately when the soft switch happens – it is not a part of the original binary. Hence, there is no need to flush any TLB or freshly load any page table. For example, in 32-bit Linux, the virtual address space was limited to 4 GB. User processes could only use the lower 3 GB, and the kernel threads could only use the upper 1 GB. Of course, there are special mechanisms for a kernel thread to read data from the user space. However, the reverse is not possible because that would be a serious security lapse. There is a similar split in 64-bit kernels as well – separate user and kernel virtual address spaces.

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 scheduling decisions are taken at the level of single threads. A thread is thus 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 work done. Linux precisely supports this notion. While switching between different threads of a thread group, it replaces only those parts of the virtual-to-physical mapping that are specific to a thread. This would include the stack region and other TLS regions (if they are defined). Here, TLS stands for *Thread Local Storage*. As opposed to replacing the complete memory map, an incremental replacement like this is much faster.

The same can be done with registers. Clearly, all the general purpose registers, the program counter and many of the ALU flags need to be stored and restored. However, there is no need to replace the CR3 register that stores the starting address of the page table. This is shared across threads. On x86 machines, any update the the CR3 register typically flushes the TLB also. Hence, this is an expensive operation that in this case can be fortunately avoided.

Finally, we need to set the `current` pointer to the `task_struct` of the new thread. Nothing more needs to be done. This is a reasonably lightweight mechanism and thus 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 can have much higher priorities. Furthermore, interrupt handlers are special because of their high priorities, restrictions, need to access native hardware and

the fact that they are independent of any user thread. Hence, the same trick of reusing the user thread and making it a kernel thread is not advisable here. There may not be any relationship between the user thread that was interrupted and the interrupt handler. Hence, in all likelihood, the interrupt handler will require a dedicated kernel thread to run. If you may recall, we had discussed the idea of an interrupt stack in Section 3.2.4, where we had mentioned that each CPU maintains a stack of interrupt threads. Whenever an interrupt arrives, we pop the stack and assign the task of running the interrupt handler to the thread that was just popped.

An advantage of doing this is that such threads would already have been initialized to have a high priority. Furthermore, interrupt handling threads have other restrictions such as they cannot hold locks or use any kind of blocking synchronization. All of these rules and restrictions can be built into them at the time of creation. 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.

3.4.3 Details of the Context Switch Process

System Call Handler

The context switch process is very specific to each architecture. This typically involves a fair amount of hardware support and the code needs to be written at the assembly level. This is because 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. Note that the `SYM_CODE_START` directive indicates that the function is written in the assembly language. Note that in this case, we use many privileged registers, which are known as model-specific registers or 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 the 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 process was saved correctly.

1. The hardware stores the program counter (`rip` register) to the register `rcx` and stores the flags register `rflags` in `r11`. Clearly, prior to 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). We shall shortly see that the `gs` segment plays a vital role in the overall process – it stores the start of the per-CPU region.

4. Almost all x86 and x86-64 processors define a special segment per 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, a system call handler stores the stack pointer (`rsp`) 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 is necessary to act as a temporary region.

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`

   (d) The code segment register

   (e) `rcx`

   (f) The rest of the general purpose registers

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

`sysret` and `iret` Instructions

The `sysret` instruction is the opposite of `syscall`. It does the reverse (returning from a system call). It transfers the contents of `rcx` 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).

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. The last value that needs to be set is that of `rip`. Setting the instruction pointer is tantamount to returning from the interrupt.

Additional Context

Along with the conventional hardware context, there are additional parts of the hardware context that needs 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 in arch/x86/include/asm/processor.h.

Every thread defines a TLS region (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.

3.4.4 The Context Switch Process

Once the context is fully saved, the user thread starts to exit 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. Especially, if there is a high-priority user thread that is waiting, then this information needs to be known at this point of time. In that case, that 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(run queue, prev task, next task)` function is defined in kernel/sched/core.c. It takes as input the run queue, which contains all the ready processes, the previous task and the next task that needs to run. Note that the scheduler has identified the next task. There are five steps in this 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`. This is the `mm_struct` associated with the `task_struct`. It primarily contains a pointer to the page table and a list of VMA (virtual memory) regions. It 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 are converted to kernel threads after a system call or interrupt. The other kind 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

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

Some relevant code is shown in Listing 3.11. If `next->mm` is NULL, which means that we are switching to a kernel thread, then 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`. If the previous thread was a user thread, then we increase its reference count. In all cases, we set the `active_mm` field to NULL of the previous thread because this information is not required any more.

This allows for a simple check. Assume a kernel to user switch. In this case, we can just compare `prev->active_mm` and `next->mm`. If both are the same, then it means that the user thread 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 thread 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.

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. The first 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 can be restored. Lastly, 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 takes care of the timing information associated with tasks. Note that tasks in the kernel maintain detailed timing information about the time that they have executed. This information is used by the scheduler. A few more memory-related adjustments are done such as adjusting the pages mapped into kernel space (known as *kmap* in Linux).

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 1 *One will often find statements of the 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 to good performance.*

Trivia 2 *One often find statements of the form:*

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

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 generating source that can be used to generate cryptographic keys.*

Chapter 4

System Calls, Interrupts, Exceptions and Signals

In this chapter, we will study the details of system calls, interrupts, exceptions and signals. The first three are the only methods to invoke the OS. Normally, the OS code lies dormant. It comes into action only after three events: system calls, interrupts and exceptions. In common parlance all three of these events are often referred to as *interrupts* even though sometimes distinctions are made such as using the terms "hardware interrupts" and "software interrupts". It is important to note that in the general interrupt-processing mechanism, an interrupt can be generated by either external hardware or internally by the program itself. Internal interrupts comprise software interrupts to effect system calls and exceptions [1] For all these mechanisms including the older way of making system calls, i.e., invoking the instruction `int 0x80` that simply generates a dummy interrupt with interrupt code 0x80, the generic interrupt processing mechanism is used. The processor even treats exceptions as a special kind of interrupts. All interrupts have their own interrupt codes; they are also known as *interrupt vectors*. An interrupt vector is used to index interrupt handler tables. Let us elaborate.

Figure 4.1 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 is ready to access the IDT using the interrupt vector as the index.

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.

[1] Note that the new *syscall* instruction for 64-bit processors, does not use this mechanism. Instead, it directly transitions the execution to the interrupt entry point after a ring level change and some basic context saving.

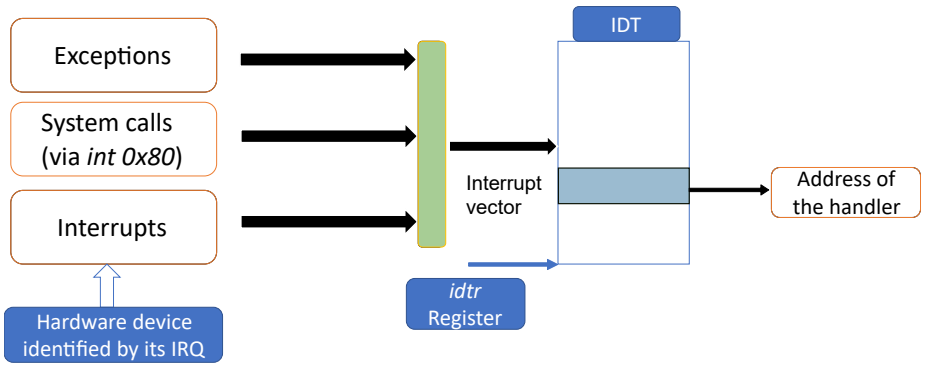


Figure 4.1: The Interrupt Descriptor Table (IDT)

4.1 System Calls

Consider the simple piece of C code shown in Listing 4.1. It shows a call to the *printf* library call. 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 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 (like ``%d'' and ``%s'') in the format string. The ellipses ... indicates a variable number of arguments.

A sequence of functions are 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` 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.

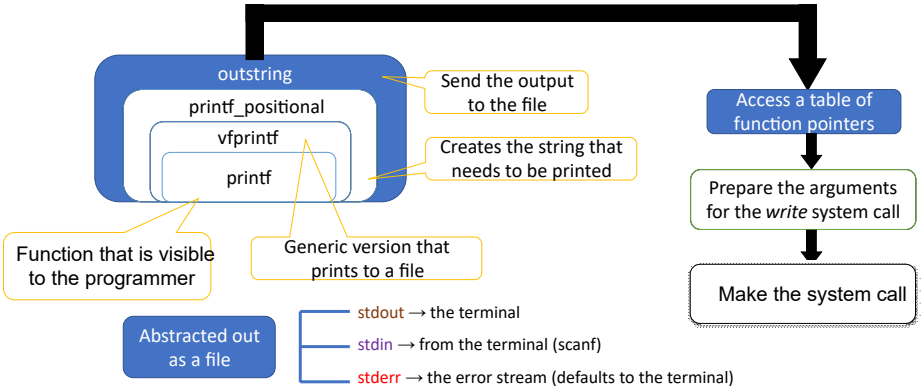


Figure 4.2: General concepts underlying library design

4.1.1 General Concepts about Underlying 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, i.e., 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 by default 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 file. Ultimately, the *write* system call is made that sends the string that needs to be printed along with other details to the OS.

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 will be induced. Hence, the context saving process cannot terminate prematurely. Saving the context is a short process and masking interrupts during this process will not create a lot of performance issues in handling even very 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 much slower as opposed to passing values via registers.

Note that a system call is a *planned activity* as opposed to an interrupt. Hence, we can free up some registers such as `rcx` and `r11` by spilling their contents. Recall that the PC 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 then subsequently called. 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.

4.1.3 Returning from a System Call

The kernel is sadly not a very gratuitous 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 routine bookkeeping, update its data structures or service devices by running kernel threads. It can also schedule other user threads.

We start 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

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

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. Other threads may create some value that is useful to the current thread. In this case also, the thread that wishes to yield the CPU gets this flag set.

If this flag is set, then the scheduler needs to run and find the most worthy process to run next. The scheduler uses very complex algorithms to decide this. The scheduler treats the TIF_NEED_RESCHED flag as a coarse-grained estimate. Nevertheless, 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 its prerogative.

The context restore mechanism follows the reverse sequence vis-a-vis the the context switch process. Note that there are some entities such as segment registers that normally need not be stored but have to be restored. The reason is that because they are not transient (*ephemeral*) in character. We don't expect them to get changed often, especially by the user task. Once, they are set, they typically continue to retain their values till the end of the execution. Their values can be stored in the task_struct. At the time of restoring a context, if the task changes, we can read the respective values from the task_struct and set the values of the segment registers.

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

Let us now discuss interrupts and exceptions. Intel processors have APIC (Advanced Programmable Interrupt Controller) chips (or circuits) that do the job of liaising with hardware and generating interrupts. These dedicated circuits are sometimes known as just interrupt controllers. There are two kinds of inter-

rupt 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 4.3 for a pictorial explanation.

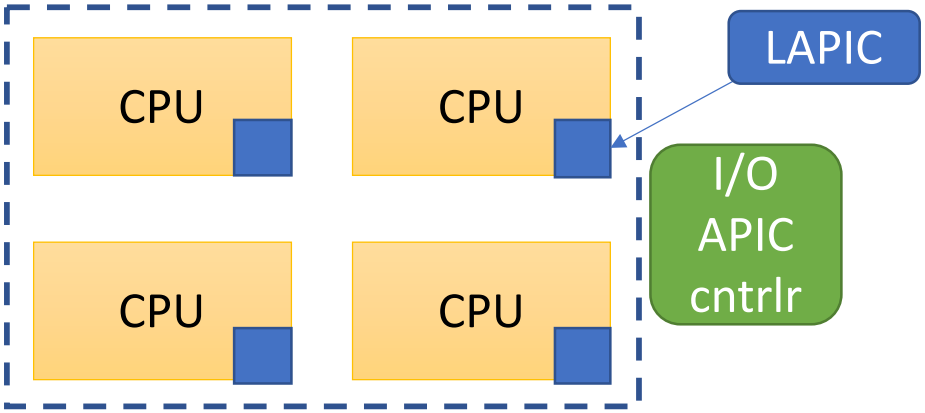


Figure 4.3: Interrupt processing mechanism in x86 processors

4.2.1 APICs

Figure 4.4 represents the flow of actions. We need to distinguish between two terms: interrupt request and interrupt number/vector. The interrupt number of 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 CPU indicating that a certain hardware needs to be serviced. There are different IRQ lines (see Figure 4.4). For example, one line may be for the keyboard, another one for the mouse, so on and so forth. In older systems, the number/index of the IRQ line was the same as the interrupt vector. However, with the advent of programmable interrupt controllers (read APICs), this has been made more flexible. The mapping can be changed dynamically. For example, we can program a new device such as a USB device to actually act as a mouse. It will generate exactly the same interrupt vector. In this way, it is possible to obfuscate a device and make it present itself as a different or somewhat altered device to software.

Here again there is a small distinction between the LAPIC and I/O APIC. The LAPIC directly generates interrupt vectors and sends them to the CPU.

The flow of actions (for the local APIC) are shown in Figure 4.4. ❶ The first step is to check if interrupts are enabled or disabled. Recall that we discussed that often there are sensitive sections in the execution of the kernel 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. Note that

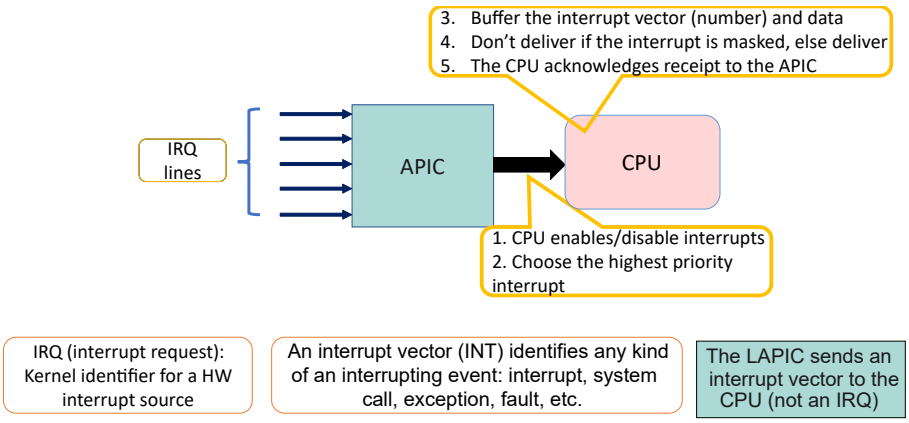


Figure 4.4: Interrupt processing flow

disabling and masking interrupts are two different concepts. Disabling is more of a sledgehammer like operation where all 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 are queued in the APIC and presented to the CPU at a later point of time when they are unmasked.

❷ Let us assume that interrupts are not disabled. Then the APIC chooses the highest priority interrupt and finds the interrupt vector for it. It also needs the corresponding data from the device. ❸ It buffers the interrupt vector and data, and then checks if the interrupt is masked or not. ❹ If it is masked, then it is added to a queue as discussed, otherwise it is delivered to the CPU. ❺ The CPU needs to acknowledge that it has successfully received the interrupt and only then does the APIC remove the interrupt from its internal queues. Let us now understand the roles of the different interrupt controllers in some more detail.

I/O APIC

In the full system, there is only one I/O APIC chip. It is typically not a part of the CPU, instead it is a chip on the motherboard. It mainly contains a redirection table. Its role is to receive interrupt requests from different devices, process them and dispatch the interrupts to different LAPICs. It is essentially an interrupt router. Most I/O APICs typically have 24 interrupt request lines. Typically, each device is assigned its IRQ number – the lower the number higher is the priority. A noteworthy mention is the timer interrupt, whose IRQ number is typically 0.

Local APIC (LAPIC)

Each LAPIC can receive an interrupt 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 seem to have any control over what is happening on CPU 1. The IPI mechanism, which is a hardware mechanism is precisely designed to facilitate this. 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 run a kernel thread on CPU 1. After doing the necessary bookkeeping steps, this kernel thread 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. It does however need the IPI mechanism to achieve this. Often, the timer chip is a part of the LAPIC. Depending upon the needs of the kernel, its interrupt frequency can be configured or even changed dynamically. We have already described the flow of actions in Figure 4.4.

Distribution of Interrupts

The next question that we need to address is how are the interrupts distributed between 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 and periodic interrupts are not generated. This behavior can change dynamically because 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 are getting 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 it has a small value. This is because any modern machine has a variety of timers. It has 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. This interrupt is originally processed

by the I/O APIC. The term "2-edge" means that this is an edge-triggered interrupt on IRQ line 2. Edge-triggered interrupts are activated when there is a level transition ($0 \rightarrow 1$ and $1 \rightarrow 0$ transitions). The handler is the generic function associated with the timer interrupt.

The "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.

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

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 later 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) that is a software-generated interrupt corresponding to a system call. 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. As we have argued many IRQs can generate this interrupt vector because there are many timers in modern systems with different resolutions. Lastly, the range 251-253 corresponds to inter-processor interrupts (IPIs). A disclaimer is due here. This is the interrupt vector range in the author's Intel i7-based system as of 2023. This in all likelihood may change in the future. Hence, a request to the reader is to treat this data as 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 quite nicely. It shows the IRQ number, interrupt vector and the hardware device. 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 – are clear from 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 can be invoked. The current 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 the same IRQ number. We can do our best to dynamically manage the IRQs such as dellocating 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 that 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, we need to check which device generated it by running all the handlers corresponding to each connected device (that share the same IRQ). These handlers will query the individual devices or inspect the data and find out. Ultimately, we will find a device that is responsible for the interrupt. This is a slow but compulsory task.

Interrupt Handling in Hardware

Let us now go to the next phase, which is in the interrupt handler circuitry of a 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 essential elements of the program state such as the $flags$ register and the PC. Here, there is a little bit of complexity. Depending upon the nature of the exception/interrupt, the program counter that should be stored can either be the current program counter or the next one. 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 PC is set to the current PC. It is assumed that the interrupt processing hardware is smart enough to figure this out. It needs to then store the next PC (appropriately computed) and the $flags$ to either known registers such as `rcx` and `r11` or on to the user's stack. This part has to be automatically done prior to starting the interrupt handler, which will be executed in software.

4.2.3 Kernel Code for Interrupt Handling

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

```
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 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 the handler, special flags and so on.

Akin to process namespaces, IRQs are subdivided into domains. This is especially necessary given that modern processors have a lot of devices and interrupt controllers. We can have a lot of IRQs, but at the end of the day, the processor will use the interrupt vector (a simple number between 0-255). It still needs to retain its meaning and be unique.

A solution similar to hierarchical namespaces is as follows: assign each interrupt controller a *domain*. 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. This is known as reverse mapping (in this specific context). Such a mapping mechanism allows us to quickly retrieve an `irq_desc` data structure given an IRQ number. Before that we need to add the interrupt controller to an IRQ domain. Typically, the `irq_domain_add` function is used to realize this. 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, we have a more nuanced solution. If there are less than 256 IRQs, then the kernel uses a simple linear list, otherwise it uses a radix tree. This gives us the best of both worlds.

The domains are organized hierarchically. We have an I/O APIC domain whose parent is a larger domain known as the *interrupt remapping domain* – its job is to virtualize the multiple I/O APICs. This 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.

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 read entries from the keyboard. Such policies can be enforced with IRQ domains.

4.2.4 Managing the IDT Table

The IDT maps the interrupt vector to the address of the handler.

The initial IDT is set up 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 emergency. Also in many cases prior to the OS booting up, the boot loader 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 reinitialize or overwrite it. For every single device and exception-generating situation, entries need to be made. These will be custom entries and only the kernel can make them because the BIOS would simply not be aware of them – they are very 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 3.2.8).

The kernel maps the IDT to the `idt_table` data structure. Each entry of this table is indexed by the interrupt vector. Each entry 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. 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. The rest of the lookup can be done in hardware and interrupt handlers can 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 you would usually expect such as a Graphics card or a network card. These devices are essential to the operation of the full system.

Next, it makes a call to `init_IRQ` to setup the per-CPU interrupt stacks (Section 3.2.4) and the basic IDT. Once that is done the LAPICs and the I/O APIC 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 4.3. The function `apic_intr_mode_init` specifically initializes the APICs on x86 machines.

Listing 4.3: The function pointers associated with IRQ handling
source : arch/x86/kernel/x86_init.c

```
.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 in this case 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 clustered in a 2-level hierarchy. This 32-bit register has 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 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 (refer to the code in `arch/x86/kernel/apic/io_apic.c`). For every single pin in the I/O APIC that is connected to a hardware device, we need to probe the device and setup an IRQ data structure for it. Next, for each I/O APIC in the system there is a need to create an IRQ domain for it and all the constituent hardware IRQs to the domain. Note that a large multi-socket system may have many I/O APICs.

4.2.5 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.

Listing 4.4: The interrupt entry point
source : arch/x86/kernel/irq.c

```
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 entry point to the IDT is shown in Listing 4.4. The `vector_irq` array is a 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 obtain the `irq_desc` data structure, we can process the interrupt by calling the `handle_irq` function. Recall that the interrupt descriptor stores a pointer to the function that is meant to handle the interrupt. The array `regs` contains the value 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. 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. Its 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 is started. It is conventionally known as the bottom half, whose job is to complete the unfinished work of interrupt handling.

Recall that an IRQ can be shared across devices. When an IRQ is set to high, we don't know which device has raised an interrupt. It is important to call all the handlers associated with the IRQ one after the other, until one of them associates the interrupt with its corresponding device. This interrupt handler

can then proceed to handle the interrupt. The rest of the interrupt handlers associated with the IRQ will return NONE, whereas the handler that handles it returns either HANDLED or WAKE_THREAD.

The structure `irq_desc` has a linked list comprising `struct irqaction*` elements. It is necessary to walk this list and find the handler that is associated with the device that raised the interrupt.

Note that all of these handlers (of type `irq_handler_t`) are function pointers. They can either be 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 or at boot time, the kernel locates the device drivers for every such device. The device driver registers a list of functions with the kernel. Whenever an interrupt arrives on a given IRQ line, it is necessary to invoke the interrupt handlers of all the devices that share that IRQ line. Basically, we traverse a list of function pointers and invoke one after the other until the interrupt is successfully handled.

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 communicate urgent data with the device. Note that such interrupt handlers are not allowed to make blocking calls or use locks. Given that they are very high priority threads, we do not want to wait for a long time to acquire a lock or potentially get into a deadlock situation. Hence, they are not allowed to use any form of blocking synchronization. Let us elaborate.

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. These are clearly attributes of ultra high priority threads that you want to 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. We schedule the work for a later point in time. The bottom half thread does not have the same restrictions of the top half thread. It can acquire locks, perform complex synchronization and can take a long time to complete. Also interrupts are enabled when a bottom half thread is running. Given that such threads have a low priority, they can execute for a long time. They will not make the system unstable.

4.2.6 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 is very different. Table 4.6 shows a list of some of the most common exceptions supported by Intel x86 processors and the latest version of the Linux kernel.

Many of the exceptions are self-explanatory. However, some need some additional explanation as well as justification. Let us consider the "Breakpoint"

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

exception. This is pretty much a user-added exception. While debugging a program using a debugger like *gdb*, we normally want the execution to stop at a given line of code. This point is known as a *breakpoint*. This is achieved as follows. 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 that indicates which line of the code corresponds to which program counter value, or which variable corresponds to which memory address is stored in typically the DWARF format. A debugger 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, then 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 encounters that program counter. Every x86 processor has dedicated debug registers (DR0 ... DR3 and 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. The debugger then lets the programmer know. Note that after the processor raises the Breakpoint exception, the program that was being debugged remains effectively paused. It is possible to analyze its state at this point of time. The state includes the values of all the variables and the memory contents.

The other exceptions correspond to erroneous conditions that should normally not arise such as accessing an invalid opcode, device or address. An important exception of consequence 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.

Exception Handling

Let us now look at exception handling (also known as *trap* handling). For every exception, we define a macro of the form (refer to Listing 4.5) –

Listing 4.5: Declaration of a trap handler
source : arch/x86/include/asm/idtentry.h

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

We are declaring a macro for division errors. It is named `exc_divide_error`. It is defined in Listing 4.6. The generic function `do_error_trap` handles all the traps (to begin with). Along with details of the trap, it takes all the CPU registers as an input.

Listing 4.6: Definition of a trap handler
source : arch/x86/kernel/traps.c

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

There are several things that an exception handler can do. The various options are shown in Figure 4.5.

The first often is clearly the most innocuous, which is to simply send a signal to the process and not take any other kernel-level action. This can for instance happen in the case of debugging, where the processor will generate an exception upon the detection of a debug event. The OS will then be informed and the OS needs to send a signal to the debugging process. This is exactly how a breakpoint or watchpoint work.

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 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.

The third option is meant to genuinely be an exceptional case. It is 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).

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 instance, 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. When the CPU will encounter an unknown instruction (in this case the cosine instruction), 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 was actually the cosine operation that the instruction was trying to compute. However, that instruction is not a part of the ISA of the current processor. In this case, it is possible to use other existing instructions and perform the computation to compute the cosine of the argument and populate the destination register with

the result. The running program can be restarted at the exact point at which it trapped. The destination register will have the correct result. It 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. However, the code now becomes completely portable. Had we not implemented this patching mechanism via exceptions, the entire program would have been rendered useless. A small performance penalty is a very small price to pay in this case.

The last option is known as the *notify_die* mechanism, which implements the classic observer pattern in software engineering.

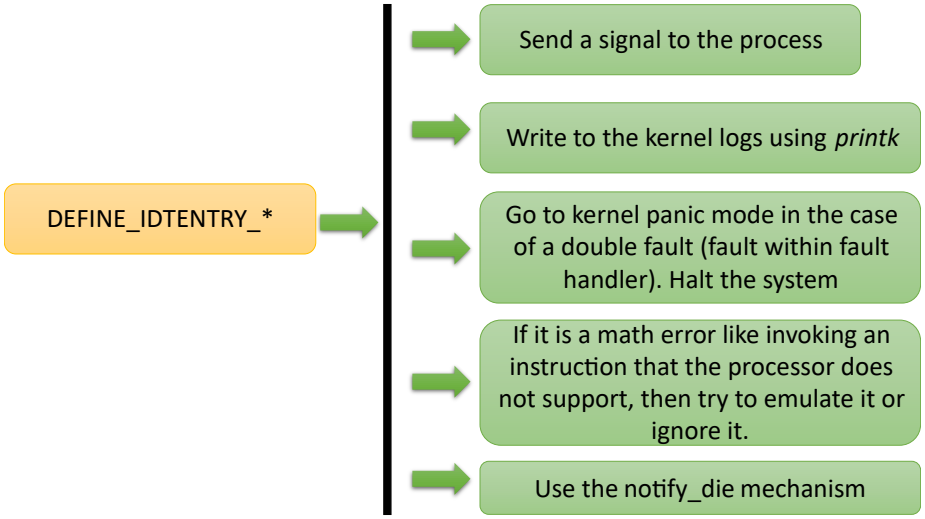


Figure 4.5: Exception handling in Linux
source : arch/x86/kernel/traps.c

The notify_die Mechanism

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), then 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 log the event. These two processes are clearly doing different things, which was the original intention. We can clearly add more processors to the chain of listeners, and do many other things.

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 invoke all the interrupt handlers that share an IRQ line in sequence. The return value indicates whether the interrupt was successfully handled or not. In that case, we would like to handle the interrupt only once. However, in the case of an exception, multiple handlers can be invoked and they can perform different kinds of processing. They may not enjoy a sense of exclusivity (as in the case of interrupts). Let us elaborate on this point by looking at the return values of exception handlers that use 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

4.3 Softirqs, Threaded IRQs, Work Queues

Let us now look at a set of bottom-half mechanisms that are used to store deferred work items. These are used by the top half and bottom/bl half interrupt handlers to synchronize amongst themselves.

Modern versions of Linux use softirqs and threaded IRQs, which are specialized bottom-half mechanisms to store and execute deferred work items later. These are not used for doing other kinds of regular processing. Linux has a more generic mechanism known as *work queues*. They 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, whereas soft and threaded IRQs were always designed for more specialized interrupt processing tasks.

A little bit of an explanation of the terminology is due 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` (represents a variable in code).

4.3.1 Softirqs

A regular interrupt's top-half handler is known as a hard IRQ. It is bound by a large number of rules and constraints regarding what it can and cannot do. A *softirq* on the other hand is a bottom half handler. There are two ways that it can be invoked (refer to Figure 4.6).

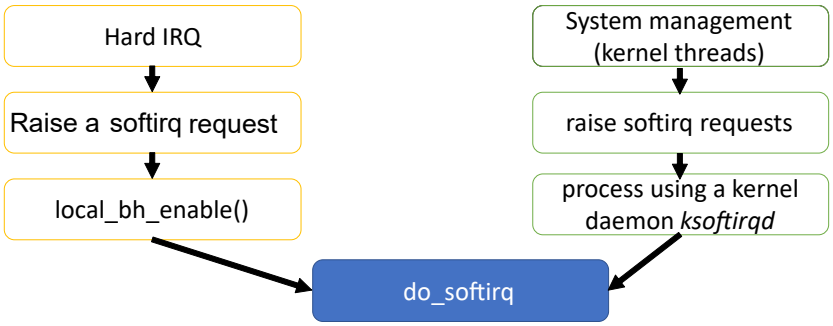


Figure 4.6: Two ways of invoking softirqs

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 later using a softirq thread. It is important to call the function `local_bh_enable` after this such that the processing of bottom-half threads like softirq threads is enabled.

Then at a later point of time the function `do_softirq` is invoked whose job is to check all the deferred work items and execute them one after the other using specialized high-priority threads.

There is another mechanism of doing this type of work (the right path in the figure). It is not necessary that top-half interrupt handlers raise softirq requests. They can be raised by regular kernel threads that want to defer some 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 immediately. Hence, a deferred work item can be created as 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. `ksoftirqd` periodically follows the same execution path and calls the function `do_softirq` where it picks an item from a softirq queue and executes a function to process it.

The net summary is that softirqs are generic mechanisms that can be used by both top-half interrupt handlers as well as specialized kernel threads; both can insert softirq requests in dedicated queues and later on processed when CPU time is available.

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.

Here are examples of some types of softirqs (defined in `include/linux/interrupt.h`): HI_SOFTIRQ, TIMER_SOFTIRQ, NET_TX_SOFTIRQ, BLOCK_SOFTIRQ, SCHED_SOFTIRQ and HRTIMER_SOFTIRQ. As the names suggest, for different kinds of interrupts, we have different kinds of softirqs defined. Of course, the list is limited and so is flexibility. However, the softirq mechanism was never meant to be very generic 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 not meant to be used by 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 that is set to 1, there is a pending softirq request. Then in a known priority order, we invoke the softirq handlers for all the bits that are set to 1. These are clearly high-priority threads and run in what is known as the *softirq interrupt context*. It is less restrictive than the regular interrupt context that top-half handlers (or hard IRQ handlers) use. Blocking calls are still not allowed. However, they can be preempted by higher priority threads that typically process interrupt top halves. They can also be preempted by softirq threads; however, they cannot be preempted by threads that correspond to regular user processes.

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 that can 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*.

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 they are not as high as interrupt-processing threads that run HW IRQ 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 4.7.

Listing 4.7: Relevant part of `struct irqaction`

```
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 `irq_handler_t` is a pointer to a function that needs to be executed to handle the interrupt – a pointer to the interrupt handler.

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 been designed for this purpose. Their structure is far more elaborate and complex as compared to the rest of the mechanisms.

Broad Overview

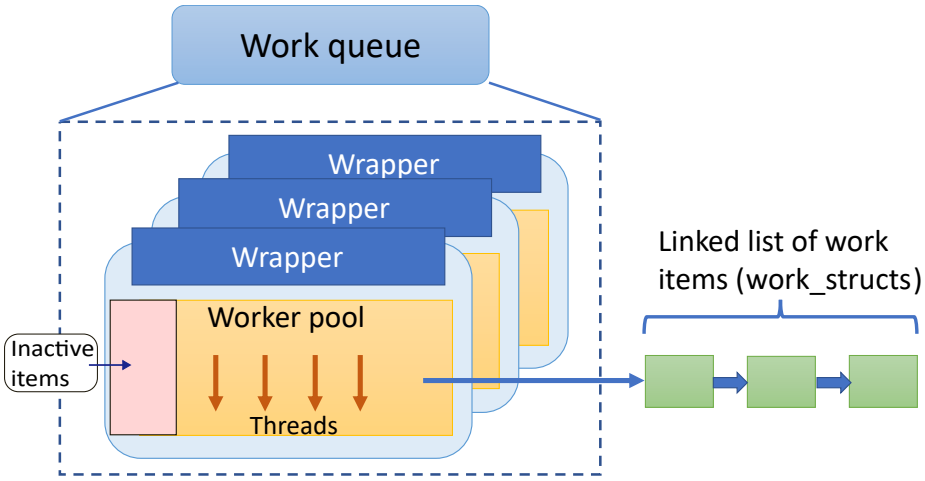


Figure 4.7: Overview of a work queue

Let us provide a brief overview of how a work queue works (refer to Figure 4.7).

A work queue is typically associated with a certain class of tasks such as high-priority tasks, batch jobs, bottom halves, etc. This is not a strict requirement, however, in terms of software engineering, this is a sensible decision.

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 set 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. 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 (referred to as a `work_struct` in the kernel code).

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.

Given that now there is an explicit size limitation, whenever there is an overflow in terms of adding additional work items, we can safely store them in the set of inactive items. When we have processed a sizeable number of active items, we can bring in work items from the inactive list into the active list. It is the role of the wrapper of a worker pool to perform this activity. Hence, there is a need to *wrap it*.

The worker pool along with its wrapper can be thought of as one cohesive unit. Now, 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.

Design of a Work Queue

The important kernel-level data structures and their relationships are shown in Figure 4.8.

A work queue is represented using the `workqueue_struct`. It points to a set of *wrapped* worker pools. A work queue needs to have at least one worker pool, which 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). Given that we are adopting a bottom-up approach here, we start with the important fields in

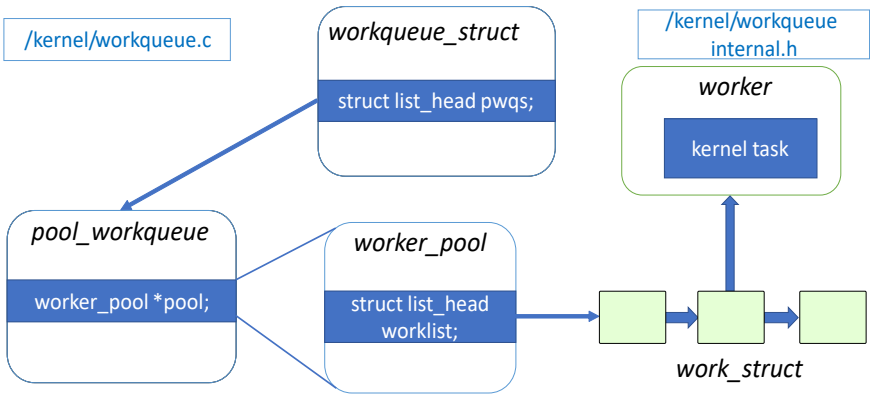


Figure 4.8: The structure of work queues

a basic work item – the `work_struct` data structure. The fields are shown in Listing 4.8.

Listing 4.8: `struct work_struct`
source : include/linux/workqueue.h

```
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
};
```

There is not much to it. 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 be a pointer as well to an object that contains all the information about the arguments. A `work_struct` represents the basic unit of work.

The advantage of work queues is that it is usable by third-party code and device drivers as well. This is quite unlike threaded IRQs and softirqs that are not usable by device drivers. Any entity can create a `struct work_struct` and insert it in a work queue. This is executed later on when the kernel has the bandwidth to execute such work.

Let us now take a deeper look at a worker pool (represented by `struct worker_pool`). Its job is to maintain a pool of worker threads that process work queue items. Whenever a thread is required it can be retrieved from the pool. There is no need to continually allocate new worker threads. We maintain a pool of threads that are pre-initialized. Recall that we discussed the notion of a pool in Section 3.2.11 and talked about its advantages in terms of eliminating the time to allocate and deallocate objects.

Every worker pool has an associated CPU that it has an affinity with, a list of `work_struct` s, a list of worker threads, and a mapping between a `work_struct` and the worker thread that it is assigned to. Whenever a new work item comes, and a corresponding `work_struct` is allocated and a worker thread is assigned

to it.

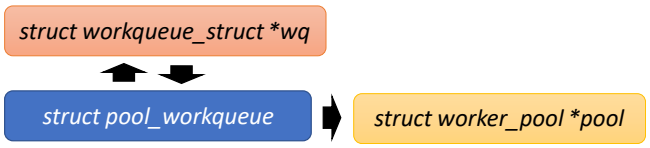


Figure 4.9: The structure of work pools

The relationship between the pool of workers, the work queue and the worker pool is shown in Figure 4.9. The apex data structure that represents the entire work queue is the `struct workqueue_struct`. It has a member, which is a wrapper class called `struct pool_workqueue`. It wraps the worker pool.

Let us explain the notion of a wrapper class. This *wrapper class* wraps the worker pool (`struct worker_pool`). This means that it intercepts every call to the worker pool, checks it and appropriately *modifies* it. Its job is to restrict the size of the pool and limit the amount of work it does at any point of time. It is not desirable to overload a worker pool with a lot of work – it will perform inefficiently. This also means that either the kernel itself is doing a lot of work or it is not distributing the work efficiently among the CPUs (different worker pools).

The standard approach used by this wrapper class is to maintain two lists: active and inactive. If there is more work than what the queue can handle, we put the additional work items in the inactive list. When the size of the pool reduces, items can be moved from the inactive list to the list of active work items.

> 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 4.9 shows examples of some common work queues defined in the kernel. As we can see they are for 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 4.9: Work queues in the system

```
extern struct workqueue_struct *system_wq;
extern struct workqueue_struct *system_highpri_wq;
extern struct workqueue_struct *system_long_wq;
extern struct workqueue_struct *system_unbound_wq;        /*
    not bound to a specific CPU */
extern struct workqueue_struct *system_freezable_wq;
   /* can be suspended and resumed */
extern struct workqueue_struct *system_power_efficient_wq;
        /* power efficient jobs */
```

```
extern struct workqueue_struct *
    system_freezable_power_efficient_wq;
```

In addition, each CPU has two 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 4.10: 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 4.10 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.

Let us assume that we did not make the call to the `exit` library function, then one of the following could have happened: if the signal blocked other signals or interrupts, then their respective handlers would be executed, if the signal was associated with process or thread termination, then the respective thread (or thread group) would be terminated (eg: SIGSEGV and SIGABRT) upon returning from the handler. If the thread was not meant to be terminated, then it resumes executing 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 turns 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

> 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. 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 multi-threaded 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 multi-threaded 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 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	They are 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 similar to SIGUSR1 – it is meant to be used by programmers in their code. They are 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 4.8: List of common signals including their definitions
source : include/uapi/asm – generic/signal.h

Please refer to Table 4.8 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 many like SIGSTOP and SIGKILL that are not sent to the processes. The kernel directly stops or kill the process, respectively.

In this sense, the SIGKILL signal is meant for all the threads of a multi-threaded process. But, as we can see, in general , a signal is meant to be handled only by a single thread in a thread group. 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. 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 process, whereas the `tgkill` call takes care of a corner case. It is possible that 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 id. This can lead to a signal being sent to the wrong thread. To guard against this the `tgkill` call takes an additional argument, the thread group id. It is unlikely that both will be recycled and still remain the same.

Regardless of the method that is used, 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 `tgkill` call is used. Sometimes, there is an arithmetic exception in a thread and thus 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 signal is blocked and such a signal arrives, 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. Also when a signal handler executes, it *blocks* the corresponding signal.

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 can lead to process termination, then the action that needs to be taken is to kill the process. Examples of such signals are SIGKILL and SIGINT (refer to Table 4.8). There can also be a case where there will be process termination but an additional file will be created called the *core dump*. 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 the 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. Here, 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 not sent to the process. They are instead intercepted by the kernel and the process is either 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. 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, process termination or stopping that thread and spawning a new thread are good options. In some cases, if the circumstances behind an exception can be changed, then the signal handler can do that like remapping a memory page or changing the value of a variable. Making such changes in a signal handler is quite risky and is only meant for black belt programmers ☺.

4.4.3 Kernel Code

Listing 4.11: Fields in the `task_struct` that pertain to signals

```
struct signal_struct    *signal;
struct sighand_struct   *sighand;
sigset_t                blocked;
sigset_t                real_blocked;    /* blocked signals
    */
struct sigpending       pending;
unsigned long           sass_ss_sp;      /* custom signal
    stack */
```

```
size_t                  sass_ss_size;
```

Let us now look at the relevant kernel code. The apex data structure in signal handling is `signal_struct` (refer to Listing 4.11). The information about the signal handler 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. 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`. Now, during the execution of any signal handler, typically more 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`.

Note the following:

$$\texttt{real\_blocked} \subset \texttt{blocked} \tag{4.1}$$

In this case we set the `blocked` signal set as 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` = `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 field, which is quite interesting. 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 in this case. 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 4.12: Fields in `signal_struct`
source : include/linux/sched/signal.h

```
struct signal_struct {
    atomic_t live;  /* number of active threads in the group
        */
    struct list_head thread_head; /* all the threads in the
        thread group */
    wait_queue_head_t  wait_chldexit; /* threads waiting on
        the wait system call */
    struct task_struct *curr_target;    /* last thread that
        received a signal */
    struct sigpending shared_pending;  /* shared list of
        pending signals in the group */
};
```

Listing 4.12 shows the important fields in the main signal-related structure `signal_struct`. It mainly contains process-related information such as the number of active threads in the thread group, linked list of 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).

Let us now look at the next data structure – the signal handler.

struct sighand_struct

Listing 4.13: Fields in `signal_struct`
source : include/linux/sched/signal.h

```
struct sighand_struct {
    refcount_t      count;
    wait_queue_head_t   signalfd_wqh;
    struct k_sigaction  action[_NSIG];
};
```

Listing 4.13 shows the wrapper of signal handlers of the entire multi-threaded process. It actually contains a lot of information in these few fields. Note that this structure is shared by all the threads in the thread group.

The first field `count` maintains the number of `task_struct` s that use this handler. 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 or the 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 and it is not the case that signals can arrive at any point of time, and then they need to be handled immediately. This notion is captured in the field `signalfd_wqh`. 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. Processes in this case, typically wait for signals to come. Hence, processes need to be woken up. At their leisure, they can check the contents of the file and process the signals accordingly.

Now, it is possible that multiple processes are waiting for something to be written to the *signalfd* file. Hence, there is a need to create a queue of waiting processes. This wait queue is the `signalfd_wqh` field.

However, the more common method of handling signals is using the regular asynchronous mechanism. All that we need to store here is an array of 64 (`_NSIG`) signal handlers. 64 is the maximum number of signal handlers that Linux on x86 supports. 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 4.14: Fields in `signal_struct`
source : include/linux/sched/signal.h

```
struct sigaction {
    __sighandler_t sa_handler;
    unsigned long sa_flags;
    sigset_t sa_mask;
};
```

The important fields of `struct sigaction` are shown in Listing 4.14. The fields are reasonably self-explanatory. `sa_handler` is the 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 masked signals.

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 and we will very soon understand why. It uses some of the tricky features of Linux linked lists, which we have very nicely steered away from up till now.

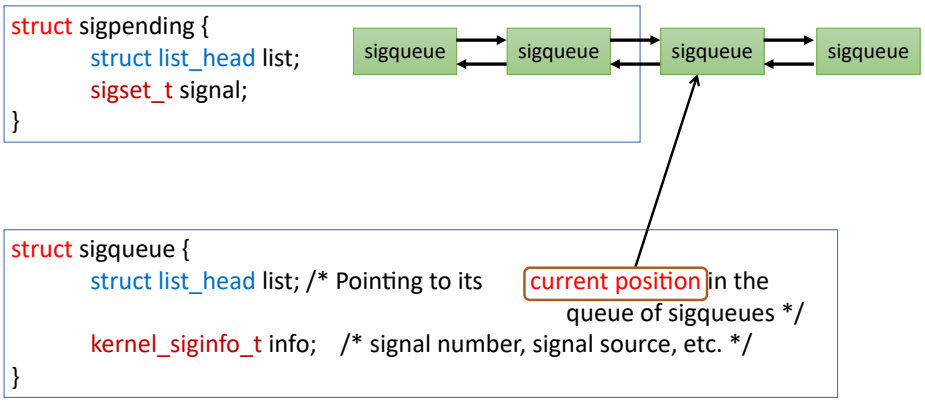


Figure 4.10: Data structures used to store pending signals

Refer to Figure 4.10. 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 i^{th} signal is pending for the process. Note that this is why there is a requirement that two signals of the same type can never be pending for the same process.

Each entry of the linked list is of type `struct sigqueue`. Note that we discussed in Appendix C that in Linux different kinds 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 is a part of a linked list, hence, it is mandated to have an element of type `struct list_head`. This points to the linked lists on the left and right (previous and next), respectively. Each entry encapsulates the signal in the `kernel_siginfo_t` structure.

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 a signal that has been raised.

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. In this case, there is no need to involve any kernel thread or kernel stack. Signal handling is completely done in user space. Hence, instead of burdening the kernel, it is a better idea to save the context in the user space stack of the signaled thread itself. The data structures to capture the context are shown in Listing 4.15.

Listing 4.15: User thread's context stored by a signal handler

```
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 process's 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 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 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 million dollar idea. Before launching the signal handler, we deliberately tweak the return address to return 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. All that we need to do is to change it to point it to a specific function, which is the `__restore_rt` function in the glibc standard library.

When the signal handler returns, it will now return and start executing the `__restore_rt` function. This function does a lot of important things. It does some bookkeeping and makes the all 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. It 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. The original thread then resumes.

Chapter 5

Synchronization and Scheduling

In this chapter we will discuss one of the most important concepts in operating systems namely synchronization and scheduling. The first 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 amongst the threads or processes to acquire the resource: this is also known as a race condition. Such data races can lead to errors. As a result, only one of the processes needs to access the shared resource at any point of time.

Once all such synchronizing conditions have been worked out, it is the role of the operating system 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 are such a wide variety of practical use cases, situations and circumstances, there is no one single universal scheduling algorithm that outperforms all the others. In fact, we shall see that for different situations, different scheduling algorithms perform very differently.

5.1 Synchronization

5.1.1 Introduction to Data Races

Consider the case of a multicore CPU. We want to do a very simple operation, which is just to 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 the register 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 5.1. 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 – execute 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 update `count` concurrently. This situation is called a *data race* (a more precise and detailed definition follows later).

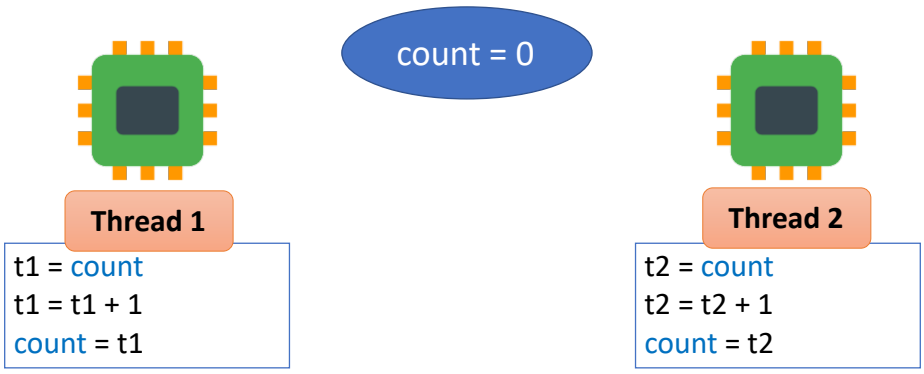


Figure 5.1: Incrementing the `count` variable in parallel (two threads). The run on two different cores. `t1` and `t2` are thread-specific variables mapped to registers

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.

❶ 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.

❷ 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 5.1 and one possible execution in Figure 5.2. Note that a parallel program can have many possible executions. We are showing one of them, which is particularly

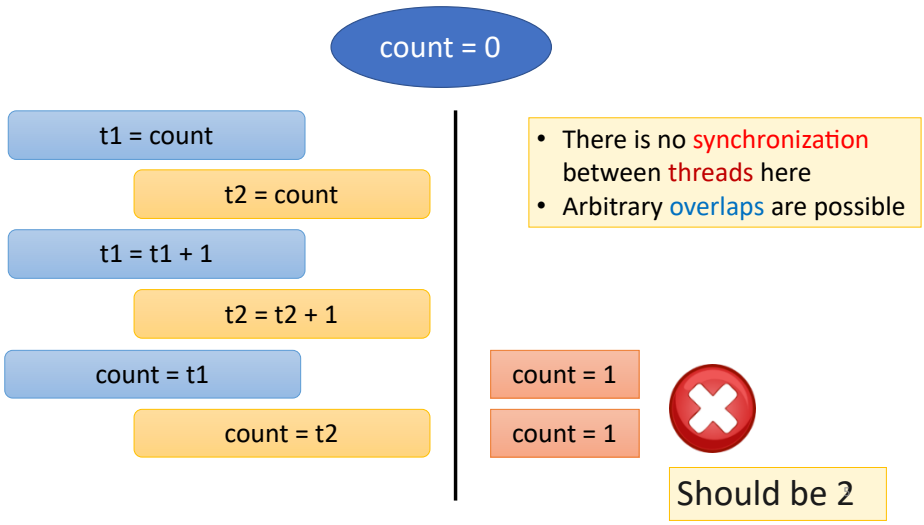


Figure 5.2: An execution that leads to the wrong value of the `count` variable

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 a register (temporary variables `t1` and `t2`, respectively). Finally they increment their respective registers and then they store the incremented value in the memory address corresponding to `count`. Since we are calling the instruction `count++` twice, we expect that the final value of `count` should be equal to 2 (recall that it is initialized to 0).

Here we get to see that the value of `count` is equal to 1, which is clearly incorrect. Basically because there was a competition or a data race between the threads, and 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. What we instead should have done is allowed one thread to complete the entire sequence of operations first, and then allowed the other thread to begin executing the sequence of instructions to increment the variable `count`.

The main issue here is that of competition or the overlapping execution, and thus there is a need for a *locking* mechanism. A lock needs to be acquired before we enter such a sequence of code, which is also referred to as a *critical section*. The idea is that we first acquire a lock, which basically means only one thread can proceed past the lock. Or in other words, if multiple threads are trying to acquire the lock at the same time, then only one of them is successful. After that, the succeeding thread proceeds to execute the instructions in the critical section, which in this case is incrementing the value of the variable `count`. Finally, there is a need to release the lock or unlock it. Once this is done, one of the other threads that has been waiting to acquire the lock can again compete for it and if it wins the lock acquiring race, it is deemed to acquire the lock. It can then begin to execute the critical section. This is how traditional code works using locks and this mechanism is extremely popular and effective – in fact, this is

the de facto standard. All such shared variables such as `count` should always be accessed using such kind of a lock-unlock mechanism. This mechanism avoids such competing situations because locks play the role of access synchronizers (see Figure 5.3).

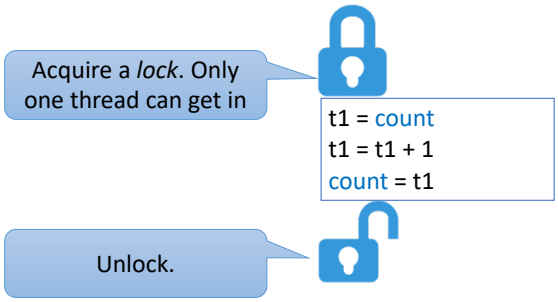


Figure 5.3: Protection of a critical section with locks

Figure 5.4 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`.

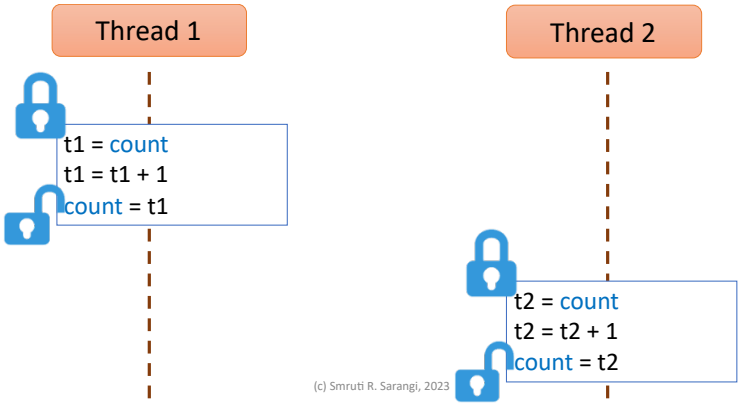


Figure 5.4: Two threads incrementing `count` by wrapping the critical section within a lock-unlock call pair

5.1.2 Design of a Simple Lock

Let us now look at the design of a simple lock (refer to Figure 5.5). 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 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 at address A (*test* phase). If the value is equal to 1, then it

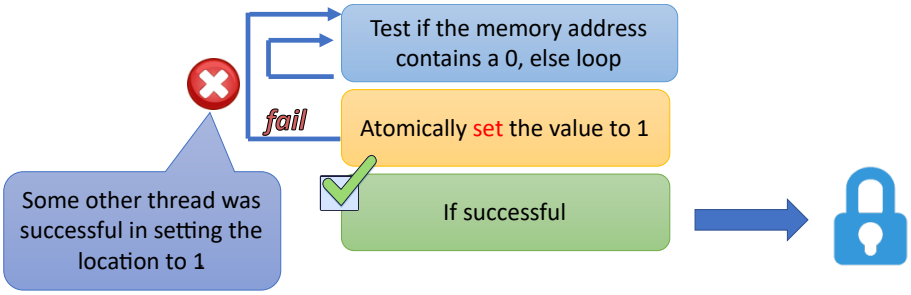


Figure 5.5: The test-and-test-and-set (TTAS) lock

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 (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 locks.

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 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 lock variable from 0 to 1, only one of them will be successful. The hardware guarantees this

The `test-and-set` instruction returns 1 if it is successful, and it returns 0 if it fails (cannot set $0 \rightarrow 1$). Clearly we can extend the argument and observe that if there are n threads that all want to convert the value of the lock variable from 0 to 1, then only one of them will be succeed. The thread that was 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 – it 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 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. All that needs to be done is that the value of the location at the lock needs to be set back to 0

(free). However, bear in mind that if one takes a computer architecture course, one will realize that this is not that simple. This is because 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 has been unlocked. This normally does not happen as architectures and compilers tend to *reorder* instructions. Also, it is possible that the instructions in the critical section are visible to other threads before a lock is fully acquired unless additional precautions are taken. This is again an unintended consequence or reordering that is done by compilers and machines for performance reasons. Such reordering needs to be checked.

The Fence Instruction

This is why 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 as well as 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 or the lock operation itself 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.

Important Point 1 *Fence instructions are expensive in terms of performance. Hence, we need to minimize them. They are however required to ensure correctness in multi-threaded 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.

5.1.3 Theory of Data Races

A *data race* is defined as a **concurrent** and **conflicting** access to a shared variable. Two accesses 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 of course 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. We need to first appreciate the notion of a happens-before relationship in concurrent systems. Event a is said to happen before event b if in a given execution, a leads to a chain of events that ultimately lead to b.

This can be seen in Figure 5.4, where we show how two threads execute two instances of the `count++` operation. The increment of the `count` variable for the first time leads to the unlock operation. There is a happens-before relationship between this unlock operation and the subsequent lock operation. This leads to the second update of the `count` variable. Therefore, we can say that there is a happens-before relationship between the first and second updates to the `count` variable. Note that this relationship is a property of a given execution. In a different execution, a different happens-before relationship may be visible. A happens-before relationship by definition is a transitive relationship.

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 $\rightarrow$ unlock operation $\rightarrow$ lock operation $\rightarrow$ reads/writes in the second critical section (so and so forth). 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 the 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.

Important Point 2 *An astute reader may argue that there have to be data races in the code to acquire the lock itself. However, those happen in a very controlled manner and 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.*

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 5.6 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 this program is not properly labeled. This is why it is important that we ensure that the same variable is protected by the same lock (could also be the same set of multiple locks).

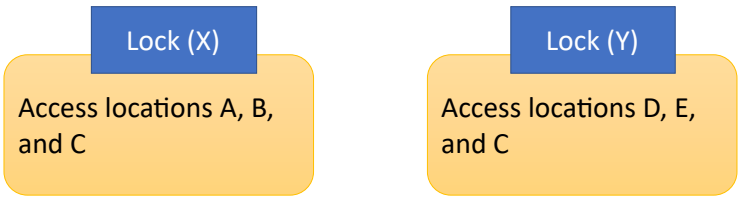


Figure 5.6: A figure showing a situation with two critical sections. The first is protected by lock X and the second is protected by lock Y. Address C is common to both the critical sections. There may be a data race on address C.

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 circular or cyclic wait. This basically means that in a deadlocked situation, no thread can make any progress. In Figure reffig:deadlock, we see such a situation with locks.

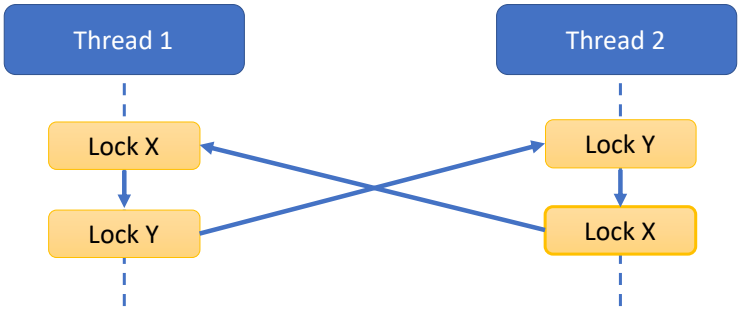


Figure 5.7: A situation with deadlocks (two threads)

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 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.

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 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 5.7, 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 5.7).

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 5.7, 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

Figure 5.8: The Dining Philosopher's problem (source: Wikipedia, Benjamin D. Eshram, licensed under CC-BY-SA 3.0)

In this context, the Dining Philosopher's problem is very important. Refer to Figure 5.8, 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 ones) and starting to eat is the same as entering the critical section. This means that both the forks have been 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 of 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 has been created. Let us look at the rest of the deadlock conditions, which are non-preeemption and hold-and-wait, respectively. Clearly mutual exclusion will always have to hold because a fork cannot be shared between neighbors at the same moment of time.

Preemption – forcibly taking away a fork from the neighbor – seems to be difficult because the neighbor can also do the same. Designing a protocol around this idea seems to be difficult. Let us try to relax hold-and-wait. A philosopher may give up after a certain point of time and put the fork that he has acquired back on the table. Again creating a protocol around this appears to be difficult because it is very easy to get into a livelock.

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 a philosopher picks the 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 n^{th} 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 and $n-1$. 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 n^{th} 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. He first picks up his right fork and then his left one. This basically means that the n^{th} philosopher has acquired both the forks and is thus eating his food. He is not waiting. We therefore do not have a deadlock situation over here.

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. We need to design a lock acquisition protocol such that it is never possible to let's say never 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. In the second phase, we release all the locks. Here the 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 a large number of 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 an ascending order of addresses.

Let us prove deadlock prevention by contradiction. Assume that there is a circular wait. Let us annotate each edge uv in this circular loop with the lock address A – Process P_u wants to acquire lock A that is currently held by P_v. As we traverse this list of locks (in the circular wait cycle), the 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 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 circular waiting loop. 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 other 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 a process or forcefully taking the locks away from it.

Starvation and Livelocks

Along with deadlocks, there are two more important issues that need to be discussed, they are *starvation* and *livelocks*. Starvation basically means that a thread tries to 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. The hardware instructions that atomically set the values of the memory locations or perform read-modify-write operations do not guarantee fairness or success in a finite number of attempts. In this sense, they do not ensure that a given thread is going to be successful in acquiring the lock.

We may need to thus wait forever to acquire a lock or acquire any kind

of shared resource. 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. Clearly *starvation freedom* is a very desirable property because it indicates that within a finite (in some cases bounded) amount of time, a thread gets access to the resource. Furthermore, this also means that starvation freedom 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 would indicate a desirable outcome.

For example, consider two people 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.

5.1.5 Pthreads and Synchronization Primitives

Let us now look at pthreads or Posix threads or *pthreads*, 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. Both the threads need to execute the `func` function: the argument is sent as a pointer and the return value is also a pointer to an object. In this case, we extract the argument (thread id) and we simply print it. The argument in this example is the pthread id. Let us now come to the `main` function that creates two pthreads. We compile it using the command `gcc prog_name -lpthread`.

Listing 5.2: Code to create two pthreads and collect their return values

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

pthread_t tid[2];
int count;
void* func(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,
                                 &func, 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);
}
```

In the `main` function, two pthreads are created. The arguments to the `pthread_create` function are a pointer to the pthread structure, a pointer to a pthread attribute structure that shall control its behavior (NULL in this example), the function pointer 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.

The return value of the `func` function is quite interesting. It is a `void *`, which is a generic pointer. In our example, it is a pointer to an integer that is equal to 2 times the thread id. When a pthread function (like `func`) returns, akin to a signal handler, it returns to the address of a special routine. Specifically, it does the job of cleaning up the state and tearing down 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 is exactly the pointer that the pthread function returns. We can proceed to dereference the pointer 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. Let us try to increment a count.

Incrementing a Shared Variable using Lock and Unlock Calls

Listing 5.3: Lock-unlock using pthreads

```
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 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.

Incrementing a Shared Variable without Using Locks

Listing 5.4: Lock-unlock using atomic fetch and add

```
#include <stdatomic.h>
atomic_int count = 0;

void * fetch_and_increment (void *arg) {
    atomic_fetch_add (&count, 1);
}
```

We use atomics, originally defined in C++ 11. These atomic operations encapsulate atomic hardware instructions. They are clearly constrained by the nature of instructions provided by the target hardware. Most processors, provide an atomic version of the fetch and add instruction that is guaranteed to complete in a bounded amount of time. Let us live with this assumption. x86 processors provide such a facility. All that needs to be done is add the `lock` prefix to an add instruction. We can thus conclude that the `atomic_fetch_add` instruction on x86 processors 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 is also a lock-free algorithm that is guaranteed to complete within a fixed number of cycles. Such algorithms are clearly way better than variants that use locks. It is just that they are very hard to code and verify if the task

that needs to be achieved is complex. Let us look at another example that uses another atomic primitive – the compare-and-swap 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. We shall refer to this method as the CAS method, henceforth. 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 unequal (`val` $\neq$ `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 try again until the CAS method is successful. Note that there is no guaranteed termination. 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, let us delve deeper into this. Let us understand the theory of concurrent non-blocking algorithms.

5.1.6 Theory of Concurrent Non-Blocking Algorithms

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 should justify the performance benefit.

Correctness of Concurrent Non-Blocking Algorithms

Let us now proceed to formally define what a non-blocking program or algorithm is. It turns out that there are a large number of non blocking algorithms; they can be classified into broadly three different classes: obstruction-free, lock-free and wait-free. 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 concurrent algorithms to operate on a shared queue. The *operations* can be *enqueue* and *dequeue*, respectively. Similarly, we can define operations on a concurrent stack. Each such operation has a *start* and *end*, respectively. They 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 akin to a write (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. The difference is quite **fundamental**.

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*. For many readers, this may appear to be non-intuitive because how can a large method that may require hundreds of instructions, appear to execute in one instant? Well, it turns out that it can "appear" to execute in an *instant*, even though it may in practice it may not be so. Let us assume that this is doable for the time being. 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.

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 an

ascending order of physical time, then we can arrange all the methods sequentially. If we think about it, this is a way of mapping a parallel execution to a sequential execution, as we can see in Figure 5.9. 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. If this sequential execution satisfies the semantics of the algorithm, then it is said to be *legal*. For example, in Figure 5.9, 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.

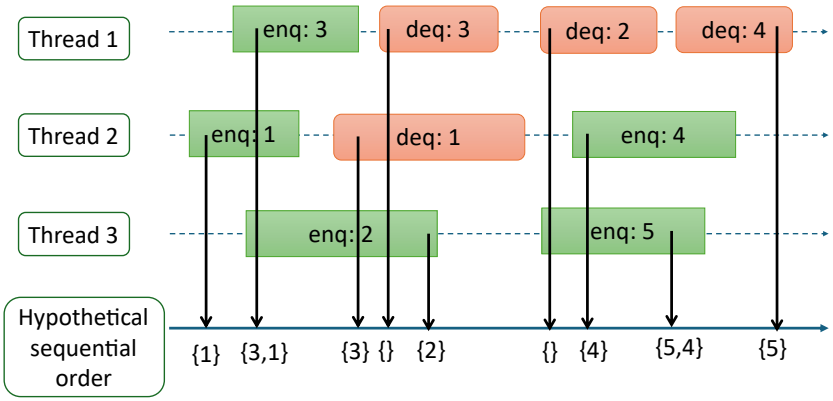


Figure 5.9: A parallel execution and its equivalent sequential execution. Every event has a distinct start time and end time. In this figure, we assume that we know the completion time. We arrange all the events in ascending order of their completion times in a hypothetical sequential order at the bottom. 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.

The key question that needs to be answered is where is this *point of completion* vis-a-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 threads. This is a very strong correctness criterion of a parallel execution. We are, of course, assuming 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. In fact, the execution shown in Figure 5.9 is linearizable.

This notion of linearizability is summarized in Definition 8. 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 would alternatively prefer to use the notion of a logical time instead. Nevertheless, let us stick to physical time for the time being.

Definition 8 *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).*

Now, 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 not important. This is why we keep using the word "appears" throughout the definitions.

Notion of Memory Models

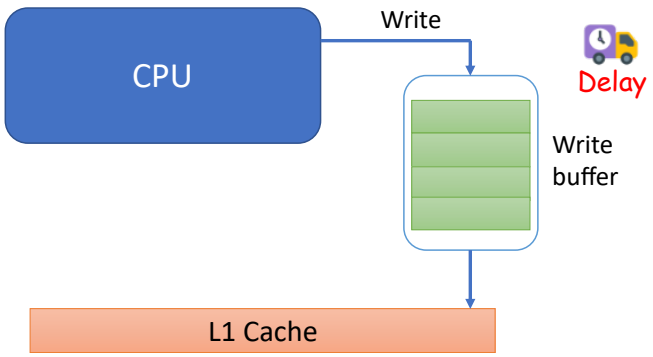


Figure 5.10: A CPU with a write buffer

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 5.10). 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, that 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 the 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 to enable a host of performance-enhancing optimizations in the compiler, processor and the memory system. These optimizations involve delaying and **reordering** instructions. As a result, the completion time can be well beyond the end time. The more relaxed we are in such matters, higher is the performance.

The question that naturally arises is how do we guarantee the correctness of algorithms? 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 an ascending order of completion times, then the sequence is *legal* – it satisfies the semantics of the concurrent system. For complex concurrent data structures like stacks and queues, linearizability is preferred; however, for simpler operations like reads and writes at the hardware level, many other models are used. These models that precisely define, which parallel executions comprising just reads and writes are legal or not, are known as *memory models* or *memory consistency models*. Every multicore processor as of today defines a memory model. It needs to be respected by the compiler and library writers.

If we just confine ourselves to reads, writes and basic atomic operations like test-and-set or compare-and-swap, then we 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.

Let us consider another memory model called sequentially consistency (SC), which again stands out in the space of memory models. It is perceived to be quite slow in practice and thus not used. However, it is used as a gold standard for correctness.

Sequential Consistency

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.

Consider the following execution. Assume that `x` and `y` are initialized to 0. They are global variables. `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, or their execution is interleaved. Regardless of the scheduling policy, we will never observe the outcome `t1 = t2 = 0` if the memory model is SC or linearizability. This 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.

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.

A modern memory model specifies a lot of rules with regards to which pairs of instructions can be reordered and also by whom: the hardware or the compiler. The 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*.

Important Point 3 *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 1 (Section 5.1.2). They can be understood better in the context of the current discussion. They basically stop reordering. A fence basically 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]). This is why creating critical sections is so important 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.

A more nuanced definition is captured in the RC (relaxed consistency) memory model. It defines two types of special fence operations: *acquire* and *release*. The *acquire* operation corresponds to *lock acquire*. It mandates that no instruction that is 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. Otherwise, it would be tantamount to disrespecting the lock. Note that an *acquire* is *weaker* than a full fence, which also specifies the ordering of operations before the fence (in program order). Similarly, the *release* operation corresponds to *lock release*. As per RC, the *release* operation can complete only if all the operations before it 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. Instead, we rely on a notion of causality between events where an event can be anything starting from a basic read or a 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. However, it is important to note that we are not relying on any notion of physical time. Instead 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 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 one basic atomic 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 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 will execute 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

It is called obstruction freedom, which basically says that in an n-thread system, if we set any set of $(n-1)$ threads to sleep, then the only thread that is active will be able to complete its execution in a bounded number of internal steps. This means that we cannot use locks because if the thread that has acquired the lock gets swapped out or goes to sleep, no other thread can complete the operation.

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 the 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 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 execute. We have already mentioned that there is no correlation between the time it takes to complete an internal step across 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 a 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 entire count increment operation. The rest of the threads will fail in that iteration. However, that is not of a great concern here because at least one thread achieves success.

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 pigeon hole 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. This is the case because we are saying that if the system as a whole takes a certain number of steps (let's say k'), then at least one thread successfully completes its operation. Now, if $n-1$ threads in the system are quiescent, then only one thread is taking steps and within k' steps it has to complete its operation. Hence, the algorithm is obstruction free.

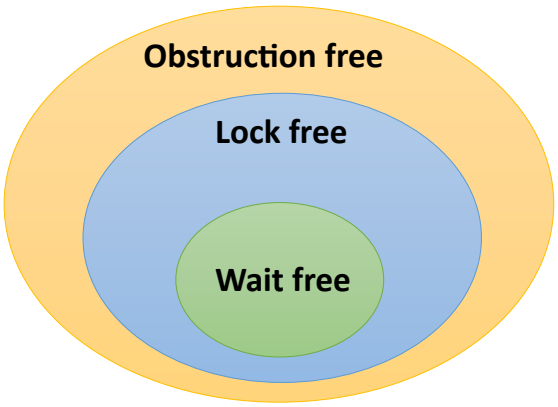


Figure 5.11: Venn diagram showing the relationship between different progress guarantees

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 5.11. All of these algorithms cannot 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, which is also provably correct. This part is correct, however, wait-free algorithms tend to be very slow and also are 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 week 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.

There is a fine point here. Many authors have replaced the *bounded* property in the definitions with *finite*. The latter property is more theoretical and often does not gel well with practical implementations. Hence, we have not decided

to use it in this book. We will continue with *bounded steps*, where the bound can be known in advance.

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 --;
```

In this case, a semaphore maintains a multi-valued `count`, which always needs to be positive. 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 inside the kernel (see Listing 5.7). Of course, it is possible to have user-level semaphores as well. However, let us only discuss kernel-level semaphores in this section.

However, if the count is not equal to 0, then the count 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 the 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, assuming that the queue is not empty. The process that was woken up is assumed to acquire the semaphore, but the count still remains 0. Otherwise, we increment the count by 1, which is basically done where there are no processes waiting on the semaphore.

We shall 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 a solution in user space that are known as condition variables.

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 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 wakeup 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.

Important Point 4 *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 smaller 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, which means that we can invoke a read method concurrently. We need a write lock that does not allow any other readers or writers to work on the queue concurrently. It just allows one writer to change the state of the queue.

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 locks 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 locks: `__rwlock` (for both readers and writers) and `__rdlock` (only for readers). The prefix `__` signifies that these are internal locks within the reader-writer lock. These locks are meant for implementing the logic of the reader-writer lock, which provides two key functionalities: get or release a read lock (allow a process to only read), and get or release a write lock (allow a process to read/write). Even though the names appear similar, the internal locks are very different from the functionality that the *composite* reader-writer lock provides, which is providing a read lock (multiple readers) and a write lock (single writer only).

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 both reads as well as writes from proceeding. This is why in the function `get_write_lock`, we wait on the lock `__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 involved, 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. 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`. Now, 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 the reference [Herlihy and Shavit, 2012].

5.1.11 Barriers and Phasers

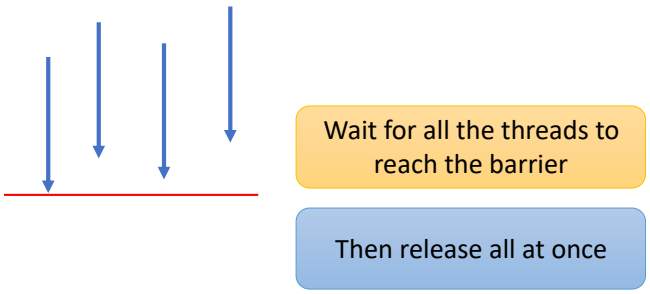


Figure 5.12: Barriers

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, any map-reduce kind of computation would typically require such rendezvous points. Let's say that we would

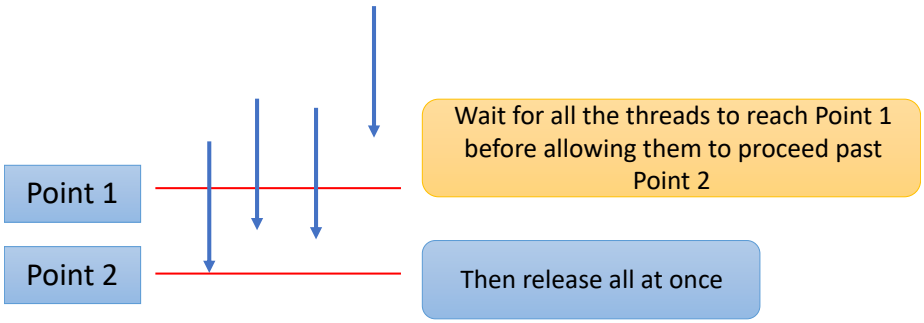


Figure 5.13: Phasers

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 and assign the i^{th} chunk to the i^{th}thread (map phase). The thread can then add all the elements in its respective chunk, and then send the computed partial sum to a pre-designated parent thread. The parent 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 optimize such operations, which 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 5.12). 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 5.13). It is somewhat uncommon, but many languages like 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 like an OS is a *queue*. All practical queues have a *bounded size*. Hence, we shall not differentiate between a queue and a queue with a maximum or bounded size. 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. There is thus a lot of benefit in implementing a concurrent queue using the aforementioned synchronization primitives.

Such concurrent queues admit multiple enqueuers and dequeuers that execute their operations in parallel. There are several options here. We can opt for a lock-free linearizable implementation, or use a version with locks. A lot of modern lock implementations are fairly fast and scalable (performance does not degrade when the number of threads increases). Let us look at different flavors of queues in this section.

Figure 5.14: A bounded queue

A conceptual view of a concurrent queue is shown in Figure 5.14, where we can observe the multiple producers and consumers.

Let us start with outlining the general convention. 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 enqueueing or dequeueing, 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.

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.

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.

Summary: We always dequeue the current head, and we always enqueue at the current tail.

5.2.1 Wait-Free Queue

Listing 5.11 shows a queue that admits just one dedicated enqueueing thread and one dedicated dequeueing thread. No other threads are allowed to use this queue and also a thread cannot change its role. This means that the enqueueing thread cannot dequeue and vice versa. Note that we shall use the code in Listing 5.11 as a running example. Most of the functions will remain the same across the implementations.

Using this restriction, it turns out that we can easily create a wait-free queue. There is no need to use any locks – operations complete within bounded time.

Listing 5.11: A simple wait-free queue with one enqueuer and one dequeuer

```
#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);
}

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, respectively, `NUM` times. 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]. 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*. They are a staple of modern versions of 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).

5.2.2 Queue with Mutexes

Let us now use the same basic template (in Listing 5.11) and create a version that allows any number of concurrent enqueuers and dequeuers. 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;
        else {
            queue[tail] = val;
            tail = INC(tail);
            status = 0;
        }
        UNLOCK(qlock);
    } while (status == -1);
    return status;
}

int deq () {
    int val;
    do {
        LOCK (qlock);
        if (tail == head) val = -1;
        else {
            val = queue[head];
            head = INC(head);
        }
        UNLOCK (qlock);
```

```
    } while (val == -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). Ultimately the lock is 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 logic 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

```
#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 multi-threaded 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 time to do this now.

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

```
#define WAIT(x) (sem_wait(& x))
#define POST(x) (sem_post(& x))
sem_t qlock, empty, full;

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. 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 waits on the semaphore, it decrements the number of free entries by 1 until the count reaches 0. After that the thread waits. Then 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.

This means that at least one free entry is available in the array. After releasing `qlock`, we signal the `full` semaphore. This indicates that an entry has been added to the queue.

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. 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 is protected by the `qlock` binary semaphore. 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 the head is equal to the 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.

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.

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 remains 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.

It is just that we are using a different set of locks for the `peak` function and the `enq`/`deq` functions. We allow multiple readers to work in parallel.

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 a large part of the address space, especially a lot of data structures. 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 within the kernel code.

Linux internally refers to a multicore processor where all the computing units are the same, as a symmetric multiprocessor (smp). The computing units have equal/similar access to memory and I/O devices.

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 are sometimes very useful, especially when they rely on atomic hardware primitives, and the code is simple to write and verify. This solves a lot of problems for us in the sense that we do not have deadlocks, and 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 lead to starvation. This is rare in practice though. We can always increase the priority of the thread that is supposedly starving. 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. Hence, a large part of the kernel still uses regular spinlocks; however, they come with a twist.

Along with being regular spinlocks that rely on busy waiting, there are a few additional restrictions. Unlike regular mutexes that are used in user space, the thread holding the lock is not allowed to go to sleep or get swapped out. 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. 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 definitely lock free. We are assuming that the lock holder will complete the critical section in a finite amount of time. 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 lock. Given that we shall 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. It is 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.

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. Given its importance in the kernel code, 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 as we can see in Listing 5.16. There is a macro called `preempt_disable`, which uses a logic similar to semaphores.

Listing 5.16: Code to enable and disable preemption
source : include/linux/preempt.h

```
#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 core idea is a preemption count variable. 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 a barrier is to ensure that the code in the critical section is not reordered and brought before the lock acquire. 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. These are synchronization operations, whereas the memory barrier is akin to a fence, which basically disables memory reordering. The preemption count is stored in a per-CPU region of memory (accessible via a segment register). Accessing it is a very fast operation and requires very few instructions.

The code for enabling preemption is shown in Listing 5.16. 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 count using an atomic operation. If the count reaches zero, it means that now preemption is allowed, so we call the schedule function. It finds a process to run on the core. An astute reader will make out that this 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.

Spinlock: Kernel Code

Listing 5.17: Wrapper of a spinlock
source : include/linux/spinlock_types_raw.h

```
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 5.17. 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 will discuss that later).

Listing 5.18: Inner workings of a spinlock
source : include/asm – generic/spin – lock.h

```
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 5.18. It is a classic ticket lock that has two components: a *ticket*, which acts like a a coupon, and the id of the next ticket (*next*). Every time that 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 number to be displayed. Once that happens, we can go to the counter at which a teller is waiting for us. The idea here is quite similar. If you think about it, you will 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 both the ticket and the *next* field. We divide the 32-bit unsigned integer into two smaller unsigned integers that are 16 bits wide. The upper 16 bits store the ticket id. The lower 16 bits store the value of the *next* field.

When a thread arrives, it tries to get a ticket. This is achieved by adding 2^{16} (1 ¡¡ 16) to the lock variable. This basically increments the ticket stored in the upper 16 bits by 1. The atomic fetch and add instruction is used to achieve this. This instruction has a built-in memory barrier as well (more about this later). Now, the original ticket can be extracted quite easily by right shifting the value returned by the fetch and add instruction by 16 positions.

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 the lower 16 bits, then we can go ahead and execute the critical section. This is easy to do using a simple typecast operation. Hear the type u16 refers to a 16-bit unsigned integer. Simply typecasting val to the type u16 type retrieves the lower 16 bits as an unsigned integer. This is all that we need to do. Then, we need to 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. Then there is a need to wait or rather there is a need to busy wait. 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. This condition checks whether the obtained ticket is equal to the *next* field in the lock variable. This macro ends up calling the macro `smp_cond_load_relaxed`, whose code is shown next.

Listing 5.19: The code for the busy-wait loop
source : include/asm – generic/barrier.h

```
#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 kernel code for the macro is shown in Listing 5.19. In this case, 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 implicitly 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.

Given that fairness is guaranteed, we will ultimately exit the infinite loop, and we will come back to the main body of the arc spinlock function. In this case, there is a need to introduce a memory barrier. Note that this is a generic pattern? Whenever we get a lock or acquire a lock, there is a need to insert a memory barrier after it. This ensures that prior to entering the critical section all the reads and writes are fully completed and are visible to all the threads in the smp system. Moreover, no instruction in the critical section can complete before the memory barrier has completed its operation. This ensures that changes made in the critical section get reflected only after the lock has been acquired.

Listing 5.20: The code for unlocking a spinlock
source : include/asm – generic/spinlock.h

```
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 5.20. 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 little bit of 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 `isenabled` (Big endian) macro. In any case at the end of this statement, the address of the *next* field is stored in the ptr variable. Next, we get the value of the ticket from the lock variable, increment it by 1, and store it in the address pointed to by ptr, which is nothing but the address of the *next* field. 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. At the very end of the unlock function, we need to execute a memory barrier known as an smp store release, which basically 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 – Slow Path

Listing 5.21: The code to try to acquire a spinlock (fast path)
source : include/asm – generic/spinlock.h

```
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 – slow path approach, is a standard mechanism to speedup the process of acquiring locks. It should not always be necessary to make the slow lock acquire call. If there is no contention, then there should be a fast method to acquire the lock. We can always fall back to the slower method if there is contention.

Listing 5.21 shows one such function in which we try to acquire the lock, and if we are not successful, then we return false. This would mean that the system automatically falls back to the traditional lock acquire function (shown in Listing 5.18). 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 5.18). 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 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 5.22: A kernel mutex
source : include/linux/mutex.h

```
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 5.22. 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, if the lock is currently acquired and held by a thread. Otherwise, its value should be 0. This neat trick will allow us to do a compare and exchange on the `owner` field in the hope of acquiring the lock quickly. We try the fast path only once. To acquire the lock, we will compare the value stored in `owner` with 0. If there is an equality then we will store a pointer to the currently running thread's `task_struct` in its place.

Otherwise, 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 enqueueing the current thread in `wait_list`, we need to acquire the spinlock `wait_lock` first.

Listing 5.23: The mutex lock operation
source : kernel/locking/mutex.c

```
void mutex_lock(struct mutex *lock)
{
    might_sleep();  /* prints a stacktrace if called in an
        atomic context (sleeping not allowed) */
    if (!__mutex_trylock_fast(lock))   /* cmpxchg on owner
        */
```

```
    __mutex_lock_slowpath(lock);
}
```

Listing 5.23 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. For example, the kernel defines an *atomic* context in which the code cannot be preempted. In this context, sleeping is 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).

Assume that the check passes and we are not in the atomic 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`.

In the slow path, we first try to acquire the spinlock, and if that is not possible then the process 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. When the lock is released, it wakes up all such sleeping processes such they can contend for the lock. The process that is successful in acquiring the spinlock `wait_lock` adds itself to `wait_list` and goes to sleep. This is done by setting its state (in general) to UNINTERRUPTIBLE.

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.

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 do much more processing. We first have to acquire the spinlock associated with the `wait_list` (list of waiting processes). Then, we remove the first entry and extract the task, `next`, from it. The task `next` needs to be woken up in the near future such that it can access the critical section. However, we are not done yet. We need to set the `owner` field to `next` such that incoming threads know that the lock is acquired by some thread, it is not free. Finally, we release the spinlock and hand over the id of the woken up task `next` to the scheduler.

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). It is in general known to be a very scalable lock that it is quite fast. It also minimizes cache line bouncing (movement of the cache line containing the lock variable across cores). The idea is that we create a linked list of nodes where the *tail* pointer points to the end of the linked list. We then add the current node (wrapper of the `current` task) to the very end of this list. This process requires two operations: make the current tail node point to the new

node (containing our task), and modify the *tail* pointer to point to the new node. Both of these operations need to execute atomically – it needs to appear that both of them executed at a single point of time, instantaneously. The MCS lock is a very classical lock and almost all texts on concurrent systems discuss its design a great detail. Hence, we shall not delve further (reference [Herlihy and Shavit, 2012]). It suffices to say that it uses complex lock-free programming and we do not perform busy waiting on a single location, instead a thread only busy waits on a Boolean variable declared within its node structure. When its predecessor in the list releases the lock, it sets this variable to false, and the current thread can then acquire the lock. This eliminates cache line bouncing to a very large extent.

There are a few more variants like 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 5.24: The kernel semaphore
source : include/linux/semaphore.h

```
struct semaphore {
    raw_spinlock_t  lock;
    unsigned int    count;
    struct list_head wait_list;
};
```

The kernel code has its version of semaphores (see Listing 5.24). It has a spin lock (`lock`), which protects 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 the user-level semaphore. After acquiring the lock, the `count` variable is either incremented or decremented. However, if the `count` variable is already zero, then it is not possible to decrement it 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 task state to RUNNING. Given that all of this is happening within the kernel, setting the task state is very easy. All of this is very hard at the user level for obvious reasons. We need a system call for everything. However, in the kernel, we do not have those 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 deadlock or not. There is thus a need to have a deadlock avoidance and possibly a recovery mechanism. Let us focus on avoidance first (refer to Section 5.1.4). Whenever a lock is acquired, a call is made to validate the lock (`lock_acquire` in kernel/locking/lockdep.c) [Molnar, 2006]. It verifies a bunch of things.

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 concurrently acquire such that the overall complexity of the kernel is limited. Next, there is a need to validate the possible lock acquisition. 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.

We define four kinds of states: 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, interrupts (irqs) would have been disabled. However, it is also possible to acquire a lock with interrupts turned on. In this case, the state will be softirq – unsafe. Here, the thread can be preempted by a softirq handler.

In any unsafe state, it is possible that the thread gets preempted and the interrupt handler runs. This interrupt handler may try to acquire the lock. 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 hardirq – safe and hardirq – unsafe analogously. These states will be used to flag potential deadlock-causing situations.

We next *validate* the chain of lock acquire calls that have been made. Check for trivial deadlocks first (fairly common in practice): $A \rightarrow B \rightarrow A$. Such trivial deadlocks are also known as *lock inversions*. Let us now use the states. No path can contain a hardirq – unsafe lock and then a hardirq – safe lock. This allows the latter call to possibly interrupt the critical section associated with the former lock. This may lead to the lock inversion deadlock.

Let us now look at the general case in which we search for cyclic (circular) waits. We need to create a global graph where each lock instance 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 is $O(V + E)$. This is quite slow. Note that we need to check for cycles before acquiring every lock.

Let us use 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.

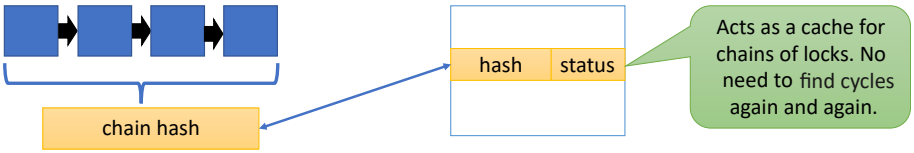


Figure 5.15: A hash table that stores an entry for every chain of lock acquisitions

Instead of checking for a deadlock on every lock acquire, we check for deadlocks far more infrequently. We consider a long sequence (chain) of locks and hash all of them. A hash table stores the "deadlock status" associated with such chains (see Figure 5.15). It is indexed with the hash of the chain. If the chain has been associated with a cyclic wait (deadlock) in the past, then the

hash table stores a 1, otherwise it stores a 0. This is a much faster mechanism for checking for deadlocks and the overheads are quite limited. Note that if no entry is found in the hash table, then either we keep building the chain and try later, or we run a cycle detection algorithm immediately. This is a generic mechanism that is used to validate spinlocks, mutexes and reader-writer locks.

5.3.5 The RCU Mechanism

5.4 Scheduling

/

Scheduling is one of the most important activities performed by an OS. 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 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 of optimality exist. Also, for a single core we may have one definition and different one for a multicore processor.

Let us introduce an optimality criterion that is widely used. It is known as 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 the time that is required to finish the entire set of jobs. In this case, there is no need to differentiate between a single core machine that can run one job at a time and a multicore machine that can run multiple jobs at a time. The notion of the makespan is the same for both the settings.

Let us now try to punch a set of holes into the definition of the makespan that we just provided. We are assuming that the time that a task 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. We also know the details of all the jobs and we have a reasonably accurate idea of how long they will take to execute. Nevertheless, for theoretical and mathematical purposes. Such assumptions are made very frequently, where we assume that the arrival time of a job and the duration of execution is known. The second point to consider is whether the tasks are preemptible or not. If they are, then the problem actually becomes much simpler. Whereas, if they aren't, then the problem becomes far more complex. Many variants of this problem with this restriction are NP-complete. This means that polynomial time algorithms for computing an optimal schedule that minimizes the makespan (or a similar metric) are not known as of today and possibly may never be known.

Along with preemptibility, we also need to look at the issue of arrival times. Some simple models of scheduling assume 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 start 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 at any point of time.

Figure 5.16 shows an example where we have a bunch of jobs that need to be scheduled. We assume that the time that a job needs to execute (processing time) is known (shown in the figure).

In Figure 5.17, we introduce an objective function, which is the mean job *completion time*. The completion time is the duration between the arrival time

Figure 5.16: Example of a set of jobs that are awaiting to be scheduled

and the time at which the job fully completes. This determines the *responsiveness* of the system. It is possible for a system to minimize the makespan yet unnecessarily delay a lot of jobs, which shall lead to an adverse mean completion time value.

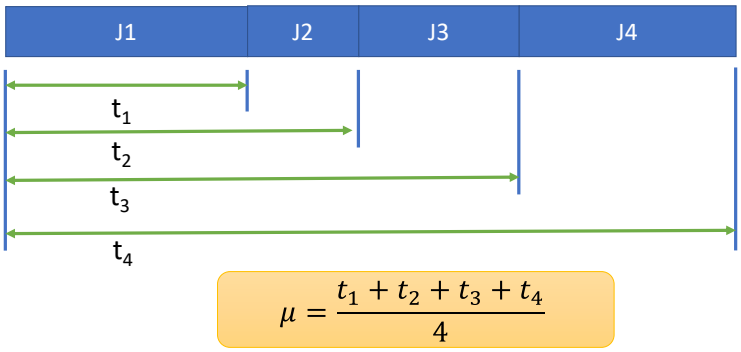


Figure 5.17: Mean completion time $\mu = \sum_i t_i/n$

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 the responsiveness).

For scheduling such a set of jobs, we have a lot of choice. 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 perform quite well. In the space of deterministic algorithms, the shortest job first (SJF) algorithm is preferred. It schedules all the jobs in ascending order of their execution times. It is a non-preemptible algorithm. We can prove that it minimizes the average

completion time.

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 problems into large classes and finds commonalities in between problems that belong to the same class. 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. It specifies the number of cores, number of jobs, and the execution 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 all the same or are different, whether the jobs have dependencies between them or whether there are job deadlines. A dependency 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. A *deadline* is thus one more type of constraint.

Finally, the last parameter is γ, which 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 in a certain sense represents a job's priority. Note that the mean completion time metric is a special case of the weighted completion time metric – all the weights are equal to 1. Let the completion time of job i be C_i. The cumulative completion time is equivalent to the mean completion time in this case because the number of jobs behaves like a constant. We can represent this criterion as ΣC_i. The makespan is represented as 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.

We will describe two kinds of algorithms in this book. We will discuss the most popular algorithms that are quite simple, and are also provably optimal in some scenarios. We will also introduce a bunch of settings where finding the optimal schedule is an NP-complete problem [Cormen et al., 2009].

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 the number of tasks is known a priori and is a constant.

The claim is that the SJF (shortest job first) algorithm is optimal in this case (example shown in Figure 5.18). Let us outline a standard approach for proving that a scheduling algorithm is optimal with respect to the criterion that

Figure 5.18: Shortest job first scheduling

is defined in the KSW problem. 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 be found in algorithm A. Assume p_j started at time t. Let us exchange jobs j and k with the rest of the schedule remaining the same. Let this new schedule be produced by another algorithm A''.

Let us evaluate the contribution to the cumulative completion time by jobs j and k in algorithm A'. It is $(t+p_j)+(t+p_j+p_k)$. Let us evaluate the contribution of these two jobs in the schedule produced by A''. It is $(t+p_k)+(t+p_j+p_k)$. Given that $p_j > p_k$, we can conclude that the schedule produced by algorithm A' is longer (higher cumulative completion time). This can never be the case because we have assumed A' to be optimal. We have a contradiction here because A'' appears to be more optimal than A', which violates our assumption.

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 $1 \mid\mid w_j C_j$ 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 a 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. Here, each job is associated with a distinct non-zero arrival time and deadline. Let us define the *lateness* as $\langle completion\_time \rangle$ - $\langle deadline \rangle$. Let us define the problem 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. The constraint 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). Jobs can 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 the *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.

If the set of jobs are *schedulable*, which means that it is possible to find a schedule such that no job misses its deadline, then the EDF algorithm will produce such a schedule. If they are not schedulable, then then the EDF algorithm will broadly minimize the time by which jobs miss their deadline (minimize L_{max}).

The proof is on similar lines and uses exchange-based arguments (refer to [Mall, 2009]).

SRTF Algorithm

Let us continue our journey and consider another problem: $1 \mid r_i, pmtn \mid \Sigma C_i$. Consider a single core machine where the jobs arrive at different times and preemption is allowed. We aim to minimize the mean/cumulative completion time.

In this case, the most optimal algorithm is *shortest remaining time first* (SRTF). For each job, we keep a tab on the time that is left for it to finish execution. We sort this list in an 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 same exchange-based argument.

Some NP-complete and Impossibility Results

Consider the case when all the jobs arrive at $t = 0$. The quality of the schedule is not affected by whether preemption is allowed or not. In does not make any difference insofar as the following optimality criteria are considered: ΣC_i or L_{max}.

Let us discuss a few NP-complete problems in this space.

- $1 \mid r_i \mid \Sigma C_i$: 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 addendum 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, this is not known.

$$t_n = \alpha t_{n-1} + \beta t_{n-2} + \gamma t_{n-3}$$

Figure 5.19: CPU and I/O bursts

Figure 5.19 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 n^{th} 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 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 like EDF, SJF and SRTF can be used. At least some degree of near-optimality can be achieved.

Let us consider the case when we have a poor prediction accuracy. We need to then rely on simple, classical and intuitive methods.

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 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 would have 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 tradeoff 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 5.20).

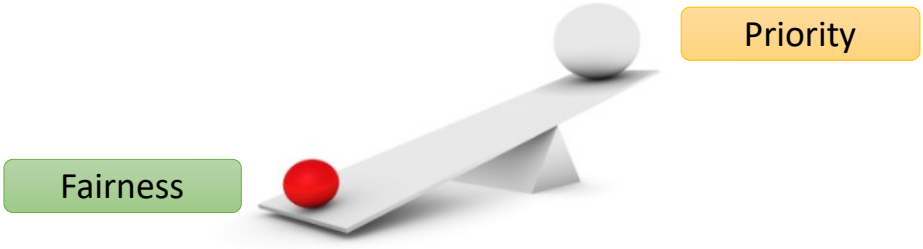


Figure 5.20: Fairness vs priority

We have discussed the notion of priorities in Linux (in Section 3.2.5). If we have a high-priority task running, it displaces other low-priority tasks. A need for fairness thus arises. Many other operating systems like Windows boost the priority of foreground processes by 3×. This again displaces background processes.

Queue-based Scheduling

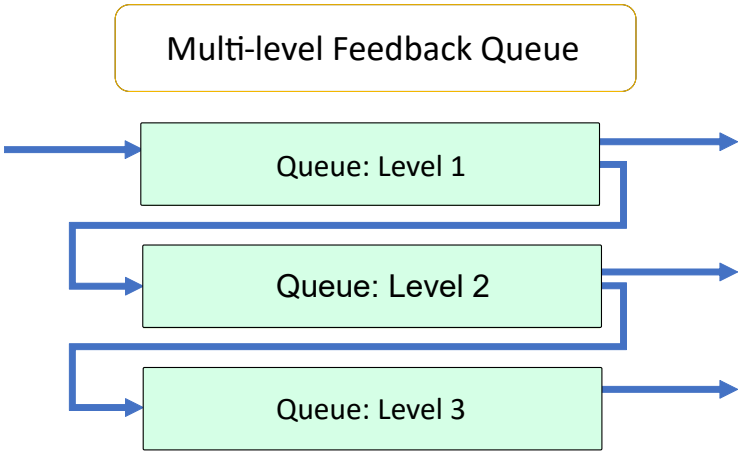


Figure 5.21: Multi-level feedback queue

A standard method of scheduling tasks that have different priorities is to use a multi-level feedback queue as shown in Figure 5.21. 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 as well as fairness.

5.4.3 Multicore Scheduling

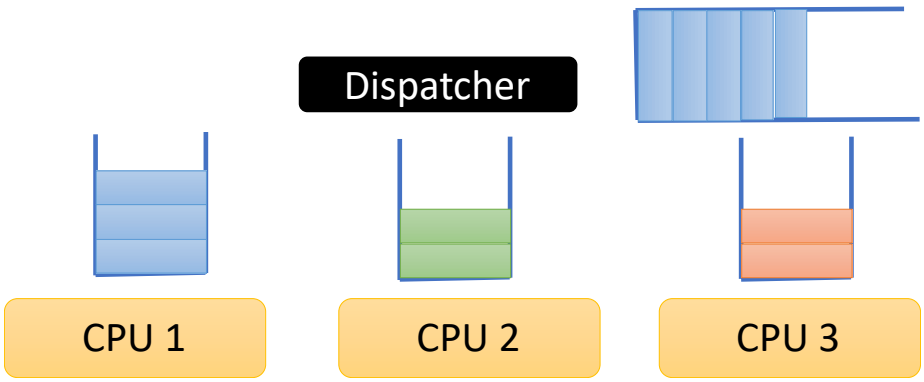


Figure 5.22: Multicore scheduling

Let us now come to the issue of multicore scheduling. The big picture is shown in Figure 5.22. We have a global queue of tasks that typically contains newly created tasks or tasks that needs 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. Let us again look at the space of problems in the multicore domain.

Theoretical Results in Multicore Scheduling

The key objective function here is to minimize the makespan C_{max}. Preemptible variants are in general easier to schedule. This is because it 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. Now, 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 is splittable arbitrarily, scheduling in this manner becomes quite simple.

However, the problem $P \mid\mid C_{max}$ is NP-complete. Jobs cannot be split and this is the source of all our difficulties.

To prove the NP-completeness of such problems, we need to map every instance of known NP-complete problems to an instance of such scheduling problems. The NP-complete problems that are typically chosen are the bin packing and partition problems. Mapping a problem's instance I to another problem's instance J, means that if we can compute a solution for J, we can adapt it to become a solution for I.

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 i^{th} 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 an optimal mapping between items to bins.

Set Partition Problem: Consider a set $\mathcal{S}$ of numbers. Find a subset whose sum is equal to a given value T (if it exists).

Once we establish a mapping from every instance of the bin packing or partition problems to a scheduling problem, we prove that the given scheduling problem in a certain sense is a superset of these known NP-complete problems. It is thus at least as hard as these problems. Given that these are provably NP-complete problems, many variants of the multicore scheduling problem are also NP-complete. Both these classical NP-complete problems assume that an item or subset cannot be split and approximations are not allowed. 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. The priority here could be the user's job priority or could be some combination of the arrival time, deadline, and the time that the job has waited for execution. When a CPU becomes free it fetches the highest priority task from the list. In case, it is not possible to execute that job, then the CPU walks down the list and finds

a job to execute. The only condition here is that we cannot return without a job if the list is non-empty. 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 often near-optimal in some cases using theoretical arguments [Graham, 1969]. Consider the problem $P \mid\mid C_{max}$. 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.

Theorem 5.4.3.1 *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 $p_1 \ldots p_n$. 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.

$$
\begin{aligned}
&\sum_{i \neq k} p_i \geq mt \\
\Rightarrow &\sum_i p_i - p_k \geq mt \\
\Rightarrow &t \leq \frac{1}{m}\sum_i p_i - \frac{p_k}{m} \\
\Rightarrow &t + p_k = C_{max} \leq \frac{\sum_i p_i}{m} - \frac{p_k}{m} + p_k \\
\Rightarrow &C_{max} \leq \frac{\sum_i p_i}{m} + p_k\left(1 - \frac{1}{m}\right)
\end{aligned}
\tag{5.2}
$$

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,

$$
\begin{aligned}
C_{max} &\leq \frac{\sum_i p_i}{m} + p_k\left(1 - \frac{1}{m}\right) \\
&\leq C^* + C^*\left(1 - \frac{1}{m}\right) \\
\Rightarrow \frac{C_{max}}{C^*} &\leq 2 - \frac{1}{m}
\end{aligned}
\tag{5.3}
$$

■

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 remain true to the basic properties of list scheduling. 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 ratios as of today for many different 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 known 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 resources. It is a very classical algorithm in this space and can be used as a basis for generating a large number of practically relevant algorithms.

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 5.23. 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 available is not as simple as finding cycles in a graph. We need a different algorithm.

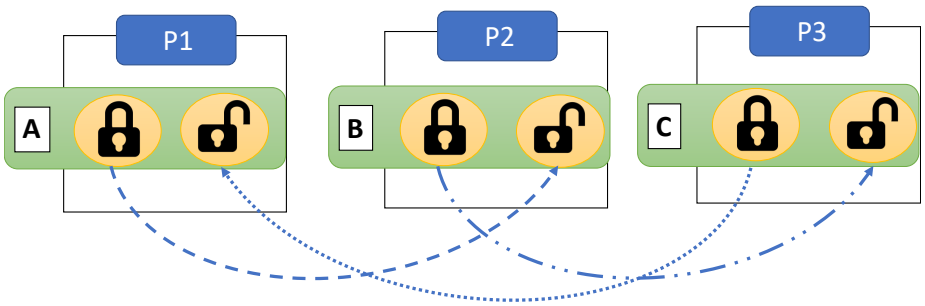


Figure 5.23: A circular dependency between processes $P1$, $P2$ and $P3$: There are three resources – A, B and C (two copies each). $P1$ has locked a unit of resource A and waits for a unit of resource B. $P2$ has locked a unit of resource B and waits for a unit of resource C. $P3$ has locked a unit of resource C and waits for a unit of resource A. There is a circular dependency across the processes and resources but because of the multiple copies, there is no deadlock. Every process gets the resource that it needs.

Let us look at the data structures used in the Banker's algorithm (see Table 5.1). There are n processes and m types of resources. The array `avlbl` stores the number of copies that we have 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 acquired `acq[i][j]` copies of resource j
`need[n][m]`	Process i may request `need[i][j]` copies of resource j. `acq + need = max`

Table 5.1: Data structures in the Banker's algorithm

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 require-

ments 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 the state is said to be unsafe. There is a need to wait till the state becomes safe again.

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: cur_cnt ← cur_cnt + acq[i]
14: done[i] ← true
15: go to find
16:
17: safety_check:
18: if ∀i, done[i] == true then
19:     return Safe
20: else
21:     return Unsafe
22: end if
```

In Algorithm 1, we first initialize the `cur_cnt` array and set it equal to `avlbl` (count of free resources). At the beginning, the request of no process is assumed to be satisfied (allotted). 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 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 i^{th} 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 `need[i]` $\preceq$ `cur_cnt`. It means that the maximum requirement of a process is less than the currently available count of resources (for all entries) – the request of process i can be satisfied.

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. It will subsequently return all the resources that it currently holds (`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**. 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, all the entries in the `done` array will be equal to **true**. It means that we are in a `safe` state – all the requests of processes can be satisfied. In other words, 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 indicates a potential deadlock.

Algorithm 2 Algorithm to request for resources

```
1:  initialize:
2:  initialize the req[] array
3:
4:  check:
5:  if need[i] ≺ req then
6:      return false
7:  else
8:      if avlbl ≺ req then
9:          wait()
10:     end if
11: end if
12:
13: allocate:
14: avlbl ← avlbl - req
15: acq[i] ← acq[i] + req
16: need[i] ← need[i] - req
```

Let us now look at the resource request algorithm (Algorithm 2). We start out with introducing a new array called `req`, which holds process i's requirements. For example, if `req[j]` is equal to k, it means that process i needs k copies of resource j.

Let us now move to the check phase. Consider the case where `need[i]` $\prec$ `req`, which basically means that every entry of `req` is greater than or equal to the corresponding entry of `need[i]`, and at least one entry is strictly greater than the corresponding entry in `need[i]`. In this case, there are clearly more requirements then what was declared a priori (stored in the `need[i]` array). Such requests cannot be satisfied. We need to return **false**. On the other hand, if `avlbl` $\prec$ `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.

Next, let us make a dummy allocation once enough resources become available (allocate). The first step is to subtract `req` from `avlbl`. This basically means that we satisfy the request for process i. The resources that it requires are not free any more. Then we add `req` to `acq[i]`, which basically means that

the said resources have been acquired. We then proceed to subtract `req` from `need[i]`. This is because at all points of time, `max=acq + need`.

After this dummy allocation, we check if the state is safe or not by invoking 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 3 Algorithm for finding deadlocks

```
1:  initialize:
2:  cur_cnt ← avlbl
3:  for ∀i do
4:      if acq[i] ≠ 0 then
5:          done[i] ← false
6:      else
7:          done[i] ← true
8:      end if
9:  end for
10:
11: find:
12: if ∃i, (done[i] == false) && reqs[i] ⪯ cur_cnt then
13:     go to update
14: else
15:     go to deadlock_check
16: end if
17:
18: update:
19: cur_cnt ← cur_cnt + acq[i]
20: done[i] ← true
21: go to find
22:
23: deadlock_check:
24: if ∀i, done[i] == true then
25:     return No Deadlock
26: else
27:     return Deadlock
28: end if
```

Let us now look at the deadlock detection algorithm (Algorithm 3). We introduce a new `reqs[n][m]` array that stores resource requests for each process. For example, `reqs[i]` stores all the resource requests for process 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**– the processes are yet to start resource acquisition. For the rest of the processes, we set `done[i]` to **true**.

Next, we find an i such that it is not done yet and `reqs`[i] $\preceq$ `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` represent the maximum number of additional requests a process can request for. Whereas, `reqs` represents the current request. We have the following relationship between them: `reqs` $\preceq$ `need`.

Let us now understand the expression `reqs[i]` $\preceq$ `cur_cnt`. This basically means that for some process i, we can satisfy its request at that point of time. We subsequently move to update, where we assume that i's request has been satisfied. Therefore, similar to the safety checking algorithm, we return the resources that i had held. We thus add `acq[i]` to `cur_cnt`. This process is done now (`done[i]` $\leftarrow$ **true**). We go back to the find procedure and keep iterating till we can satisfy the requests of as many processes as possible. When this is not possible any more, we jump to deadlock_check.

Now, if `done[i]` == **true** for all processes, then it means that we were able to satisfy the requests of all processes. 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. 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 a 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 is the `schedule` function (refer to Listing 5.25). We first try to see if we can dispatch work to other threads that possibly run on different CPUs. The kernel has a set of threads known as *kworker* threads. They perform the bulk of the kernel's work including handling deferred work and work queues. This function wakes up worker threads and assigns them some work. This is especially important if the current task is going to get blocked. 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. After the schedule function returns, we update the status of worker threads (`sched_update_worker`). This function performs simple bookkeeping.

Listing 5.25: The `schedule` function
source : kernel/sched/core.c
Also refer to [de Olivera, 2018]

```
void schedule(void)
{
    struct task_struct *tsk = current;

    /* Dispatch work to other kernel threads */
    sched_submit_work(tsk);

    do {
        preempt_disable();
        __schedule(SM_NONE);
        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. On similar lines, 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.

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 5.2). 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 class of jobs that need to be scheduled by a specific type of scheduler. Linux defines a 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.

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 60-Hz video, the deadline is 16.66 ms.

RT These are regular real-time threads that are typically used for processing interrupts (top or bottom halves), for example softIRQs.

Fair This is the default scheduler that the current version of the kernel uses (v6.2). It ensures a degree of fairness among tasks where even the lowest priority task gets some CPU time.

Idle This scheduler runs the idle process, which means it basically accounts for the time in which the CPU is not executing anything – it is idle.

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`. This is a generic structure that simply defines a set of function pointers (see Listing 5.26). 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 defines some function 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. For example, if we are implementing a deadline scheduler, then all the functions shown in Listing 5.26 point to the corresponding functions defined for the deadline scheduler. In this manner, we can create a generic scheduler that provides a common, standardized interface.

Listing 5.26: List of important functions in `struct sched_class`
source : kernel/sched/sched.h

```
/* 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 5.26, 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 structs 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 then perform several operations on it 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. Based on the scheduler's task selection algorithm an appropriate choice is made. The `pick_task` function is the fastpath that finds the highest priority task (all tasks are assumed to be separate), whereas the `pick_next_task` function is on the slowpath. The slowpath 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 between processes in a cgroup. cgroups further can be grouped into hierarchies. 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 5.2).

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 5.2: List of schedulers in Linux

The runqueue

Let us now take a deeper look at a runqueue (`struct rq`) in Listing 5.27. 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 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 (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` (`prev_mm`). The task that is chosen to execute is stored in `struct`

*core_pick.

Listing 5.27: The runqueue
source : kernel/sched/sched.h

```
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 slowpath. 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.

For regular non-real-time processes, here is the algorithm that is followed. If there are no currently executing tasks on the core in the CFS class or above, then we find the next task using the CFS scheduler (*fair*). Otherwise, 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. Subsequently, we effect a context switch.

Scheduling-related Statistics

Listing 5.28: Scheduling-related fields in the `task_struct`
source : include/linux/sched.h

```
/* 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 5.28). The key structure that embodies these statistics are variants of the `struct sched_entity` class (one for each scheduler type). For example, for CFS scheduling the `sched_entity` structure contains some pieces of information such as the time at which execution started, sum of execution time, the virtual runtime (relevant to CFS scheduling), the number of migrations, etc. The Linux `time` command gets its information structures of this type.

The `task_struct` contains a pointer to the scheduling class, which we need to have because every task is associated with a `sched_class`.

The `core_node` and `core_cookie` fields in `task_struct` have both performance and security implications. The `core_node` refers to the core that is the "home core" of the process. This means that by default the process is scheduled on `core_node` and all of its memory is allocated in close proximity to `core_node`. This of course is valid in a NUMA (non-uniform memory access) machine, where the notion of proximity 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 attack 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 a 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 an iota of execution time in a scheduling period – there is no starvation. The `sched_entity` class maintains all scheduling-related information for CFS tasks (refer to Listing 5.29).

struct sched_entity

Listing 5.29: `struct sched_entity`
source : include/linux/sched.h

```
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 enter nodes in a linked list by having a member in the structure 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 philosophy 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 5.30: weight as a function of the nice value
source : kernel/sched/core.c

```
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,
};
```

Equation shows the relation between the actual runtime and the vruntime. The vruntime is $\delta_{vruntime}$ times the actual runtime. The formula for $\delta_{vruntime}$ is

shown in Equation 5.4. Let δ be equal to the time interval between the current time and the time at which the current started executing. If *vruntime* is equal to δ then it means that we are not using a scaling factor. The scaling factor is equal to the weight associated with a *nice* value of 0 divided by the weight associated with the real nice value. We clearly expect the ratio to be less than 1 for high-priority tasks and be less than 1 for low-priority tasks.

$$\delta_{vruntime} = \delta \times \frac{weight(nice = 0)}{weight(nice)} \tag{5.4}$$

Listing 5.30 shows the mapping between nice values as 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 1.25×. For example, if the nice value is 5, the weight is 335. $\delta_{vruntime} = 3.05\delta$. Clearly, we have an exponential decrease in the weight as we modify the nice value. For a nice value if n, the weight is roughly $1024 \times (1.25)^n$. The highest priority user task has a weight equal to 88761 (86.7×). 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. 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. Please refer to the code snippet shown in Listing 5.31. If the number of runnable tasks are more than N (limit of 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 5.31: Implementation of scheduling quanta in CFS
source : kernel/sched/fair.c

```
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`).

$$slice_i = SP \times \frac{weight(task_i)}{\sum_j weight(task_j)} \tag{5.5}$$

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 unlikely case where `nr_running` $>$ `N`), then each slice is equal to G.

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 is shown in Listing 5.32. Here, if the CFS queue is non-empty, we compute the time for which a task has already executed (`ran`). If `slice` $>$ `ran`, then we execute the task for `slice` $-$ `ran` time units by setting the timer accordingly, otherwise we reschedule the current task.

Listing 5.32: `hrtick_start_fair`
source : kernel/sched/fair.c

```
if (rq->cfs.h_nr_running > 1) {
    u64 slice = sched_slice(cfs_rq, se);
    u64 ran = se->sum_exec_runtime - se->
        prev_sum_exec_runtime;
    s64 delta = slice - ran;

    if (delta < 0) {
        if (task_current(rq, p))
            resched_curr(rq);
        return;
    }
    hrtick_start(rq, delta);
}
```

Clearly, once a task has exhausted its slice, its vruntime has increased and its position needs to be adjusted in the RB tree. In any case, every time we need to schedule a task, we find the task with the least vruntime in the RB tree and *check* if it has exhausted its time slice or not. If it has, then we mark it as a candidate for rescheduling (if there is spare time left in the current scheduling period) and move to the next task in the RB tree with the second-smallest vruntime. If that also has exhausted its scheduling slice or is not ready for some reason, then we move to the third-smallest, and so on.

Once all tasks are done, we try to execute tasks that are rescheduled, and then start the next scheduling period.

Some Important Caveats

Every user has an incentive to increase its execution time. They will thus try to game the scheduler. We need to take special care for new tasks, tasks waking

up after a long sleep and tasks getting migrated. 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 there. 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 a `sched_entity` that is enqueued in the runqueue.

3. If an old task is being restored or a new task is being added, then set $\texttt{se}->\texttt{vruntime} += \texttt{cfs\_rq}->\texttt{min}_{\texttt{v}}\texttt{runtime}$. This ensures that some degree of a level playing field is being maintained.

4. This ensures that other existing tasks have a fair chance of getting scheduled

5. Always ensure that all vruntimes monotonically increase (in the `cfs_rq` and `sched_entity` structures).

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.

7. Upon a task migration or block/unblock, subtract $\texttt{cfs\_rq}->\texttt{vruntime}$ from the source `rq` and add the corresponding number for the destination `rq` (while enqueueing back again in the same or a different `rq`). This does not give an unfair advantage or disadvantage to migrating tasks.

8. Treat a group of threads (part of a multi-threaded process) or 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 either creating more threads or spawning new processes.

Calculating the CPU Load

Computing the load average on a CPU is important for taking 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, the 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 5.6.

$$load_{avg} = u_0 + u_1 \times y + u_2 \times y^2 + \ldots \tag{5.6}$$

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 Real-Time and Deadline 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}. If the set of jobs are schedulable (can complete before their deadlines), then EDF will find the schedule. 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 deadline.

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 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. The scheduling is clearly not fair. There is no notion of fairness across real-time priorities.

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.

Chapter 6

The Memory System

Managing the memory is one of the most functions of an operating system. Of course, there is a need to isolate different processes. However, after addressing correctness concerns, concerns regarding efficiency dominate. We need to ensure that the page replacement algorithm is as optimal as possible. Finding candidate pages for replacement requires maintaining a lot of data structures and doing a lot of bookkeeping. There is a need to move a lot of this work off the critical path.

We shall then appreciate that kernel memory allocation 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. We also need to understand that most of the correctness as well as security problems emanate from memory management, hence, managing memory correctly is essential.

6.1 Traditional Heuristics for Page Allocation

6.1.1 Base-Limit Scheme

Let us consider traditional heuristics for memory management. Let us go back to the era that did not have virtual memory or consider systems that do not have virtual memory such as small embedded devices. Clearly the need to isolate processes address spaces is there. This is achieved with the help of two registers known as *base* and *limit*, respectively. As we can see in Figure 6.1, 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 out of bounds.

We can visualize the memory space as a sequence of contiguously allocated regions (see Figure 6.1). There are *holes* between the allocated regions. If a new process is created, then its 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. 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 9 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 refers to a base-limit based addressing system where space in the memory system is kept empty in the form of holes.*

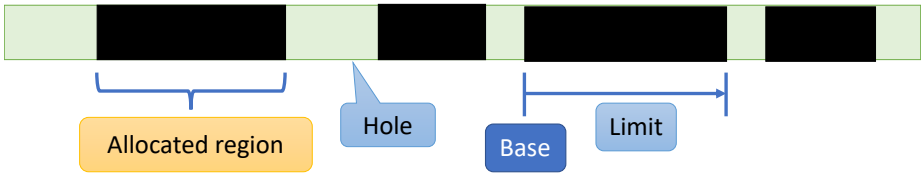


Figure 6.1: Memory allocation with base and limit registers

The next question that we need to answer is that if we are starting a new process and we are exactly aware of the maximum amount of memory that it requires, then which hole do we select for allocating its memory? Clearly the size of the hole needs to be more than the amount of requested memory. However, 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 very well 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 particularly in anticipation of the future. There are several heuristics in this space. Let us say that we need R bytes.

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 quite suboptimal. It is also possible to prove that they are

optimal in some cases assuming some simple distribution of memory request sizes in the future. In general, we do not know how much memory a process is going to access. Hence, declaring the amount of memory that a process requires upfront is quite difficult. 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 still 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 straight forward. To keep track of the free frames 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 Emde Boas tree (see Appendix C). In $log(N)$ time, We can find the first free frame and allocated it to a process. Even freeing frames is quite easy in such a system that uses a bit vector and Emde Boas tree. Internal fragmentation is a minor problem given that there is a bound on the amount of space that is wasted (limited to 4 KB).

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 hard disk or flash drive, which 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 6.2). We conceptually organize all the physical pages (frames) in a stack. Whenever a page is accessed, it is removed from the stack and placed at the top of the stack. The distance between the top of the stack and the point in the stack where a page was found is known as the *stack distance*.

Definition 10 *We maintain a stack of all pages that are present in main memory. 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. This is a standard tool to evaluate temporal locality in computer systems from caches to paging systems.*

The stack distance typically has a distribution that is similar to the one shown in Figure 6.3. Note that we have deliberately not shown the units of the x and y axes because the aim was to just show the shape of the figure and not

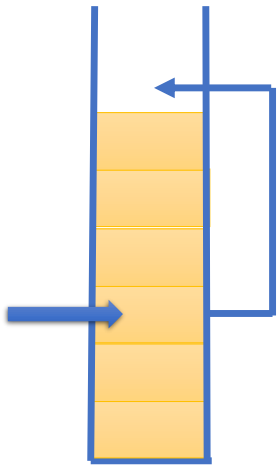


Figure 6.2: Notion of the stack distance

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 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. 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 6.3. There are a lot of distributions that have heavy tail. 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 distribution is

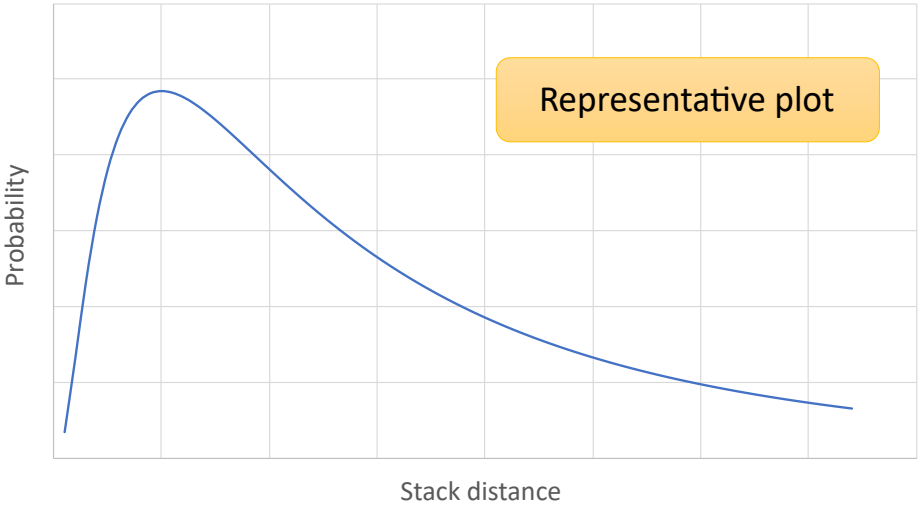


Figure 6.3: Representative plot of the stack distance

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 is a general family of algorithms that show well-defined behavior, as we shall see. Note that the stack here is a conceptual or mathematical entity. We don't actually have a stack in hardware.

We organize all the pages as a stack. Some of them are in physical memory, some of them are yet to be accessed/initialized and some of them are in the swap space. We consider the universal set of pages. These pages are ordered in descending order of replacement priority. Assume that the size of the memory is n frames (physical pages). We assume that the top n pages are stored in main memory at all times and the rest are in the swap space or have not been accessed yet. This is known as the *stack property*.

If a page is accessed, then its priority can be arbitrarily increased or decreased. This means that it needs to be then moved to a location corresponding to its new priority on the stack (possibly somewhere in the middle).

While computing the stack distance, this is exactly what we were doing. The pages were organized by their last access time. Let us assume that the priority is inversely proportional to the time that the page was last accessed. In this case, once a page is accessed, its priority becomes the maximum and it is moved to the stack top. Let us now focus on an optimal stack-based algorithm.

Optimal Page Replacement Algorithm

Similar to scheduling, we have an optimal algorithm for page replacement as well. 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 as well, where we assumed that we knew the exact time of execution of each job.

We start out with ordering all the pages in a decreasing order of the "next use" time. This basically means that we know when a page is going to be used next. Then for replacement, we choose that candidate page that is going to be used or accessed the last. We can use a priority queue to find the task with the maximum next use time. The task at the head of the queue is the candidate for replacement. The proof technique is quite similar to what we would use to prove that a scheduling algorithm is optimal. We can use a contradiction-based technique and use exchange-based arguments to prove optimality – guarantees the lowest page fault rate.

This can be thought of to be a stack-based algorithm. The priority is inversely proportional to the next-use time. Whenever a page is accessed, its next-use time is computed given that we know the future, then the page is moved to its appropriate position in the stack.

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).

Least Recently Used (LRU) Algorithm

Let us now discuss a more practical algorithm known as the least recently used algorithm (LRU). We need to 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-used 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. We also cannot encumber every memory access with computing and storing a timestamp. To find the least timestamp, these numbers have to be stored in a priority queue. Entering an entry into a priority queue requires $O(log(N))$ time. 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 but the computational overheads will still be there. Given that we need to set the timestamp whenever an access is made, maintaining such accurate

information will still be prohibitive – we are basically adding a costly timestamp update to every memory access. 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 LRU information needs to be an *infrequent operation*.

Let us leverage the page protection bits that are a part of the page table as well as the TLB. The hardware already provides them. 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 LRU information 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 useful.

Using this mechanism, when we access a page, the hardware will 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.

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. So 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 transitioned from 0 to 1. This means that it was accessed and there was a page fault. Later on the 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. Lower 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

Let us now implement the approximation of the LRU protocol. A simple implementation algorithm is known as the WS-Clock page replacement algorithm, which is shown in Figure 6.4. 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. That is set to either 0 or 1 and is stored along with the corresponding page table entry. A

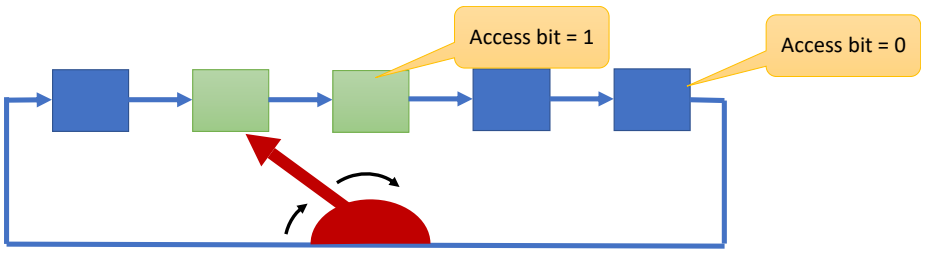


Figure 6.4: The WS-Clock algorithm

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, then it is set to 0 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 check the access bit and if it is set to 1, we reset it to 0. However, if the access bit is equal to 0, then we select that page for replacement. For the time being, the process stops at that point. Next time the pointer starts from the same point and keeps traversing the list of pages towards the end until it wraps around 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 we prioritize unmodified pages 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 result combination of bits is now $\langle 0, 0 \rangle$; we move forward. We are basically giving the page a second chance in this case as well because it was accessed in the recent past.

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 this is clearly a frequently used frame that gets written to and thus it should not be evicted or downgraded (access bit set to 0).

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. It is also known as a *soft page fault*. They however 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 the time at which 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*. 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]. Let us understand it better by looking at the two examples shown in Figures6.5 and 6.6. In Figure 6.5, 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.

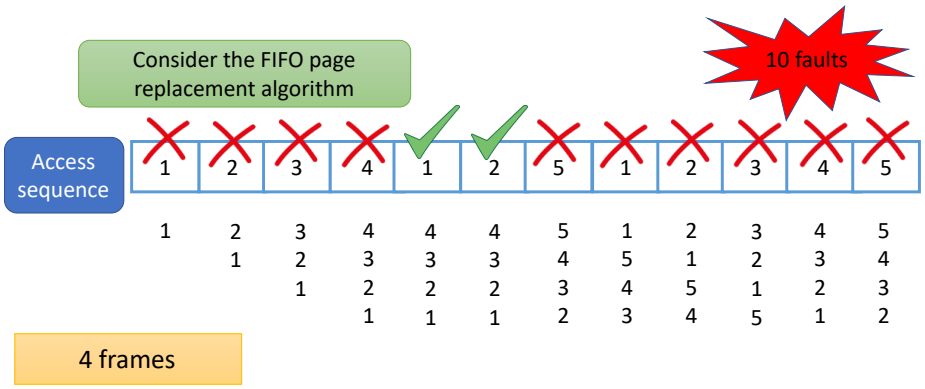


Figure 6.5: FIFO algorithm with memory capacity equal to 4 frames

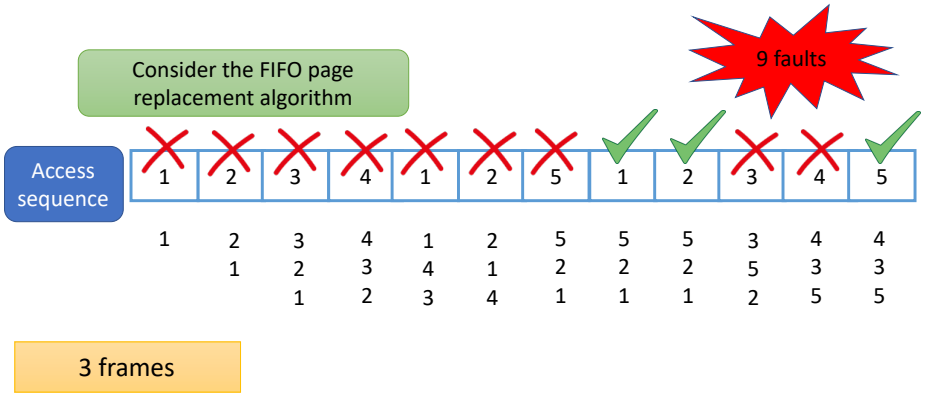


Figure 6.6: FIFO algorithm with memory capacity equal to 3 frames

In this particular example shown in Figure 6.5, 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 6.6), 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 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 are 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 can 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 to occur to a limited extent. This significantly reduces the performance of the system because page faults are associated with massive overheads.

Let us now summarize our discussion. A pure stack-based algorithm that uses a stack to arrange the pages in memory and then use the stack distance as a basis for making replacement decisions is quite impractical. For every access, we need to perform a fair amount of computation and update the stack. This algorithm is thus quite slow in practice. Hence, we need to work on approximations. In line with this philosophy, we introduced the WS-Clock and the WS-Clock Second Chance algorithms. The FIFO and a simple random replacement algorithm are in comparison much easier to implement. However, they exhibit the classic Belady's anomaly, which is clearly not in the best interest of performance and page fault reduction. The worst case performance 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.

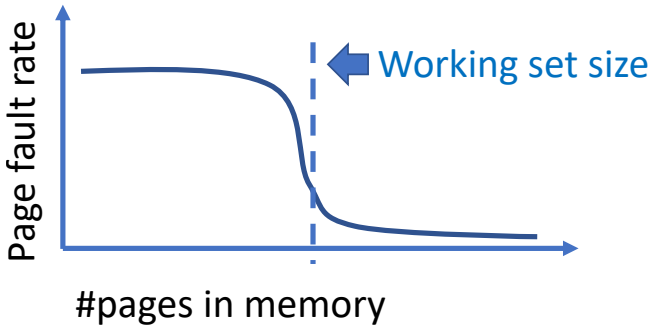


Figure 6.7: Page fault rate versus the working set size

There is a different way of answering this question. It is scientifically more reasonable. Consider the graph shown in Figure 6.7. The x axis is the number of pages that we have in memory and the y axis is the page fault rate. We 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. All the pages that we have in memory at the threshold point is the working set.

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 6.7 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.

Thrashing

Consider a system with a lot of processes. If the space that is allocated to a process is less than the size of its working set, then the process will suffer from a high page fault rate. Most of its time will be spent in fetching its working set and servicing page faults. The CPU performance counters will sadly indicate that there is a low CPU utilization. The CPU utilization will be low primarily because most of the time is going in I/O: servicing page faults. However, the kernel's load calculator will observe that the CPU load is low. Recall that we had computed the CPU load in Section 5.4.6 (Equation 5.6) using a similar logic.

Given that the load average is below a certain threshold, the kernel will try to spawn more processes to increase the average CPU utilization. This will actually exacerbate the problem and make it even worse. Now the memory that is available to a given process will further reduce.

Alternatively, the kernel may try to migrate processes to the current CPU that is showing reduced activity. Of course, here we are assuming a non-uniform memory access machine (NUMA machine), where a part of the physical memory is "close" to the given CPU. This proximate memory will now be shared between many more processes.

In both cases, we are increasing the pressure on memory. Process will spend most of their time in fetching their working set into memory – the system will thus become quite slow and unresponsive. This process can continue and become a vicious cycle. In the extreme case, this will lead to a system crash because key kernel threads will not be able to finish their work on time.

This phenomenon is known as *thrashing*. Almost all modern operating systems have a lot of counters and methods to detect thrashing. The only practical

solution to prevent thrashing is to actually reduce the number of active processes such that all of them can complete.

All of us would have experienced this while booting up a machine where we are the source of the problem. When an operating system starts booting up, it is quite slow. Our patience tends to run out because we cannot put up with the slow booting process. To address our boredom, we we click a few buttons to start a few applications in the hope of doing something and keeping ourselves busy. This exacerbates the problem and induces thrashing. We see that the system becomes even more unresponsive. Then our patience runs out again and we start a few more applications !!! Ultimately it takes a very long time for the system to actually come up and become fully functional and responsive. Sometimes because of this behavior an OS may crash as well. The right thing to do here is to actually do nothing ☺. We will then avoid thrashing. We should actually go and drink a cup of water.

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 of the 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`). Linux 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

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 are strictly separated. Let us consider the case of a 57-bit virtual addressing system. In this case, the total virtual memory is 128 PB. We partition the virtual memory space into two parts: 64 PB for a 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 6.8). Note that the figure is not drawn to scale – we have only shown some of the important regions and 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).

We have a 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 many a time, especially while dealing with external entities such as I/O devices, the DMA controller and the booting system. It is also used to store the page tables. This is a reasonably large area That allows for

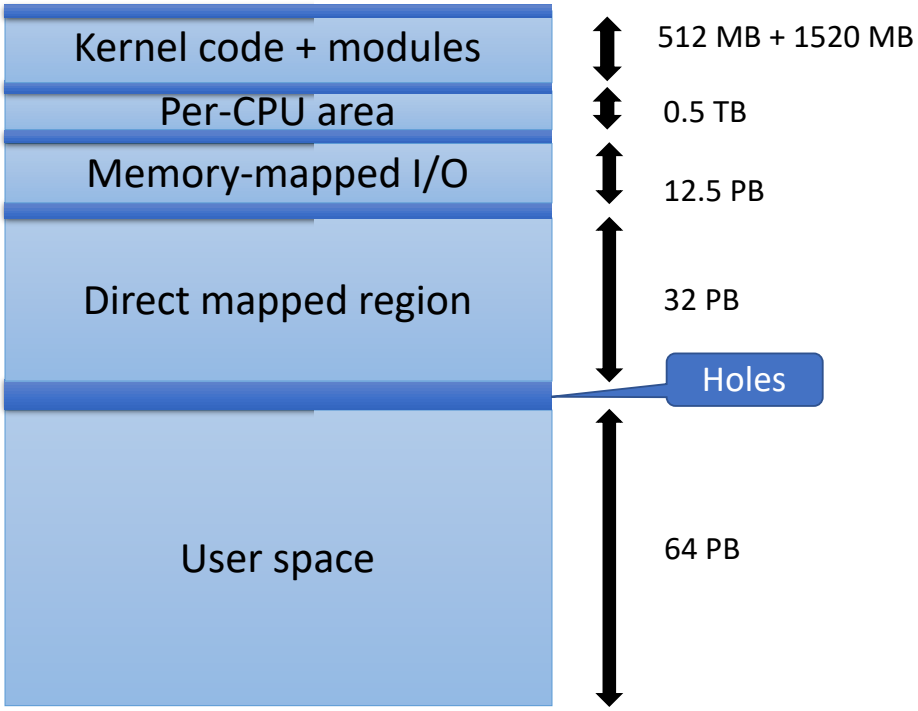


Figure 6.8: The overall virtual memory map (kernel + user)
source : Documentation/x86/x86_64/mm.rst

efficient memory accesses especially if we need large chunk of contiguous memory given the ease of translating memory addresses mapped to this region. The latency of servicing TLB misses and address translation reduce considerably.

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. Often the use of direct-mapped memory is clubbed with a hardware-supported mechanism known as *huge pages*. In this mechanism, the page size is set to a reasonably large value such as 2 MB or even 1 GB – this reduces the number of TLB misses and the translation overheads significantly.

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 the `current task` information, part of the context and preemption-related flags.

The core kernel code per se is quite small. We only reserve 512 GB for the kernel code, which again is on the higher side. It is important to note that a large part of the overall kernel code comprises the code of the device drivers. 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.

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

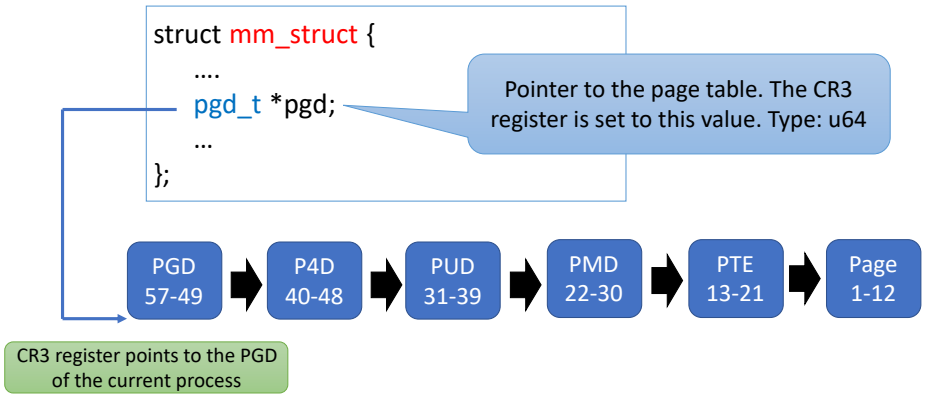


Figure 6.9: The high-level organization of the page table (57-bit address)

Figure 6.9 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 6.9. We observe that there are five levels in a page table. The highest level of the page table is known as the page directory (PGD). Its starting addresses 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. This is very expensive. Hence, various kinds of optimizations have been proposed.

We shall quickly see 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.

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 last 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.

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

Page Table Entry

Additionally, each page table entry also contains the permission bits for the page (see Table 6.3). This information is also kept in the TLB such that the hardware can check if a given operation is allowed on data stored in the page. For example, if we are not allowed to execute code within the page, then the execute permission will not be given. This is very important from a security perspective. Similarly, for code pages, the 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.

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 6.1: The follow_pte function (assume the entry exists)
source : mm/memory.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 6.1 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. It is locked prior to returning a pointer to the page table entry. This allows us to make changes to the page table entry. It needs to be subsequently unlocked after it has been used/modified by another function.

Let us now look slightly deeper into the code that looks up a table in the 5-level page table. A representative example for traversing the PUD table is shown in Listing 6.2. Recall that the PUD table contains entries that point to PMD tables. Let us thus traverse the PUD table. We find the index of the PMD entry using the function pmd_index and add it to the base address of the PUD table. This gives us a pointer to the PMD table. Recall that each entry of the PUD table contains a pointer to a PMD table (pmd_t *). Let us elaborate.

Listing 6.2: Accessing the page table at the PMD level

```
/* 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 31-39. This is achieved by shifting the address to the right by by 30 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 PUD page table that is obtained using the `pud_pgtable` function. Note that the multiplication is implicit primarily because the return type of the `pud_pgtable` function is `pmd_t *`.

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 possibly 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 the starting point of this region or some point within this region (depending upon the architecture). Note the 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. The `pud_valpud` returns the bits corresponding to the physical address of the PUD 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. This is the address corresponding to the PUD, which is assumed to be aligned with a page boundary. This physical address is then converted to a virtual address using the `__va` function. We then add the PMD index index to it 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 along 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 have 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 encapsulate 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 back to the pool of pages), its reference count is equal to zero. It is important to note that the `page` structure is ubiquitously used and that too for a large number of purposes, hence it needs to have a very flexible structure. This is where using a union with the 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 *compound* or *aggregate* page that comprises two or more contiguous pages. 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 is identified by a pointer to the head page (first page). It is a single entity insofar as the rest of the kernel code is concerned. This in itself is a very useful concept because in a sense we are grouping contiguous virtual memory pages based on some notion of application-level similarity.

Now if the first page of the folio is accessed, then in all likelihood the rest of the pages will also be accessed very soon. Hence, it makes a lot of sense to prefetch these pages to memory in anticipation of 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 to 1 GB. 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. If 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. 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.

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 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 6.3 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 6.3: Converting the page frame number to the `struct page` and vice versa

source : include/asm – generic/memory_model.h

```
#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 6.3 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).

6.2.4 Managing the TLB

TLB Design

Let us know now look at TLB-specific optimizations. Note that 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 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 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 and 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 (48^{th} or 57^{th} 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 back 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 of the 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 a 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 of the 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. Intel also 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.

Lazy TLB Mode

Let us now consider the case of multi-threaded 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 a large number of 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".

Important Point 5 *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. However, this cannot happen in the current implementation of the kernel. This is because 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 out 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/P-CID changes. 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 multi-threaded 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 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 nonuniformity in the main memory access latency, which is something that the OS needs to leverage for guaranteeing good performance.

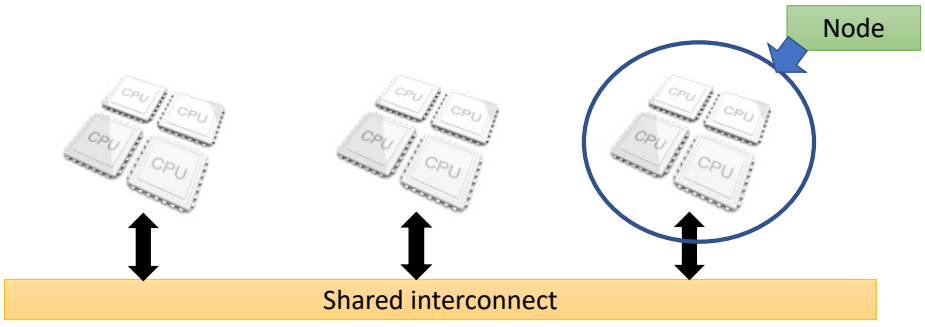


Figure 6.10: NUMA machine

Refer to Figure 6.10 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 *hierarchically*. The local memory needs to be the lowest level and the next level should comprise pointers to remote memory.

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 nonoverlapping sets.

Each such set is referred to as a *zone*. The 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 6.4 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 the 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 layout 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 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 because now the hardware can manage the pages in NVMs directly. They are all 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.

Important Point 6 *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 processor 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 6.4: The list of zones
source : include/linux/mmzone.h

```
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 6.3, 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 6.11). 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 noncontiguous. It is thus a good idea to break a noncontiguous 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 6.3 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.

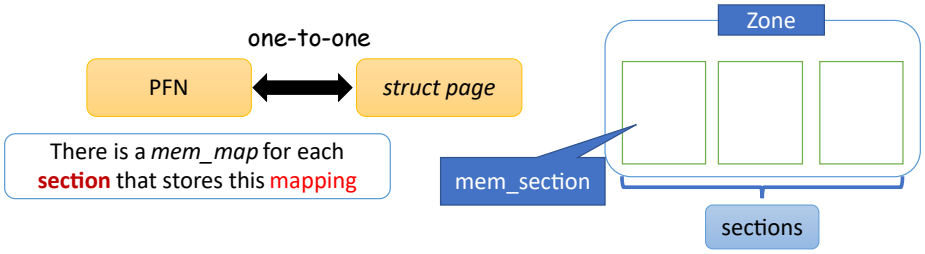


Figure 6.11: Zones and sections

Detailed Structure of a Zone

Given the short digression, let us look at the zone structure (`struct zone`) in Listing 6.5 again. 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 6.5: `struct zone`
source : include/linux/mmzone.h

```
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 6.6). `struct pglist_data` stores the relevant details. A NUMA node is identified by its node ID (`node_id`). Now, 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 less frequently used pages and migrates them to the swap space. This frees up much-needed memory space. In addition, 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 of freeing up memory and balancing 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 **??**.

Listing 6.6: `struct pglist_data`
source : include/linux/mmzone.h

```
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 in the Kernel

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 virtual 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 are 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: *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).

The concept of a buddy is shown in Figure 6.12. In this case, we consider a region of memory whose size in bytes or kilobytes is a power of 2. For example, in Figure 6.12, 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.

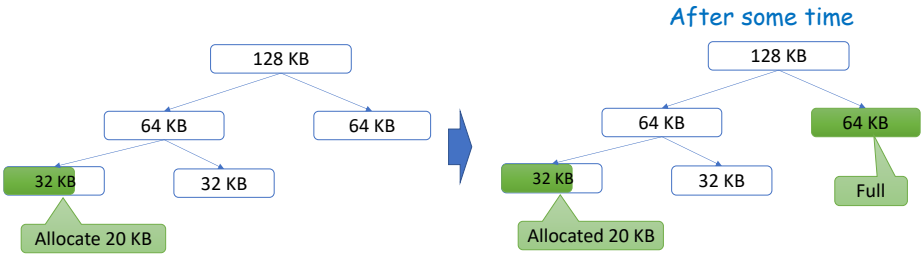


Figure 6.12: Buddy allocation

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 sub-region 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 6.12, we allocate the remaining 64 KB region (right child of the parent) to the request.

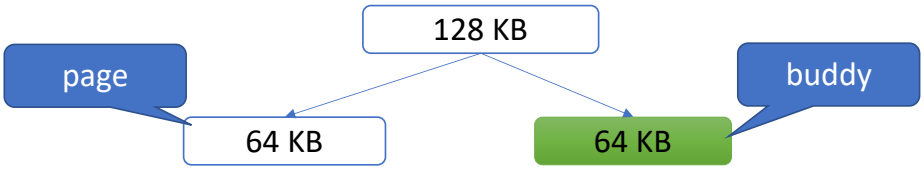


Figure 6.13: Freeing the 20 KB region allocated earlier

Let us now free the 20 KB region that was allocated earlier (see Figure 6.13). 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 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. Let us now look at the implementation. Let us refer to the region represented by each node in the buddy tree as a *block*.

Implementation

Let us look at the implementation of the buddy allocator by revisiting the `free_area` array in `struct zone` (refer to Section 6.2.5). Let us 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 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 pages, where each page is actually an aggregate page that may point to a N contiguous 4 KB pages, where N is a power of 2.

The buddy tree is thus represented by an array of linked lists – `struct free_area free_area[MAX_ORDER]`. Refer to Listing 6.7, where each `struct free_area` is a linked list of nodes (of the same order). The root's order is limited to `MAX_ORDER - 1`. In each `free_area` structure, the member `nr_free` refers to the number of free blocks (=number of pages in the associated linked list).

Note that there is a little bit of a twist 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 6.7: `struct free_area`
srcinclude/linux/mmzone.h

```
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 page. This buddy tree is an integral part of a zone. It is its default memory allocator.

This can be visualized in Figure 6.14, where we see that a zone has a pointer to a single `free_area` (for a given order), and this `free_area` structure 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

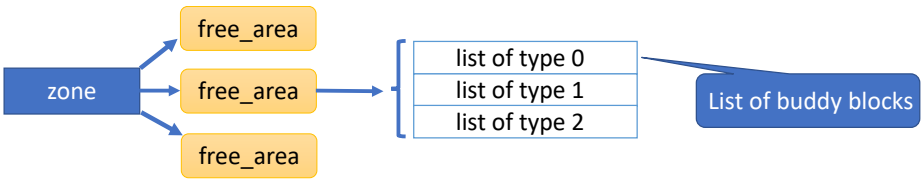


Figure 6.14: Buddies within a zone. The *type* refers to the migration type

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

Listing 6.8 shows the relevant kernel code where we traverse the list of `free_area` s 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 6.8: Traversing the list of `free_area` s in the function `__rmqueue_smallest`
srcmm/page_alloc.c

```
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;
    del_page_from_free_list(page, zone, current_order);
    ...
    return page;
}
```

Listing 6.9 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 the page, we find the page frame number of the buddy. If the buddy is not free then the `find_buddy_page_pfn` returns NULL. Then, we exit the *for* loop and go to label `done_merging`. 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 buddies of each other. We then have $A = B \oplus 2^{\phi}$. Here $\oplus$ stands for the XOR operator. Then if we coalesce them, 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 #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 6.9, where A and B are `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 6.9: Code for freeing a page
srcmm/page_alloc.c

```
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);
}
```

Once we combine a page and its buddy, we increment the order and try to combine the parent with its buddy and so on. 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.

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 in a leaf node that exceeds the minimum page size, the allocated memory always exceeds 50% of the capacity of that node (otherwise we would have split that node).

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 3.2.11. 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 situations, 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 6.15. 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 i^{th} bit is set if the i^{th} object has been allocated and some task is using it. A slab uses a `freelist` to store the indices of free objects. Every slab has a pointer to a slab cache (`kmem_cache`) that contains a large number of slabs. It manages the object pool across all the cores.

Note that all of these entities such as a slab and the 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.

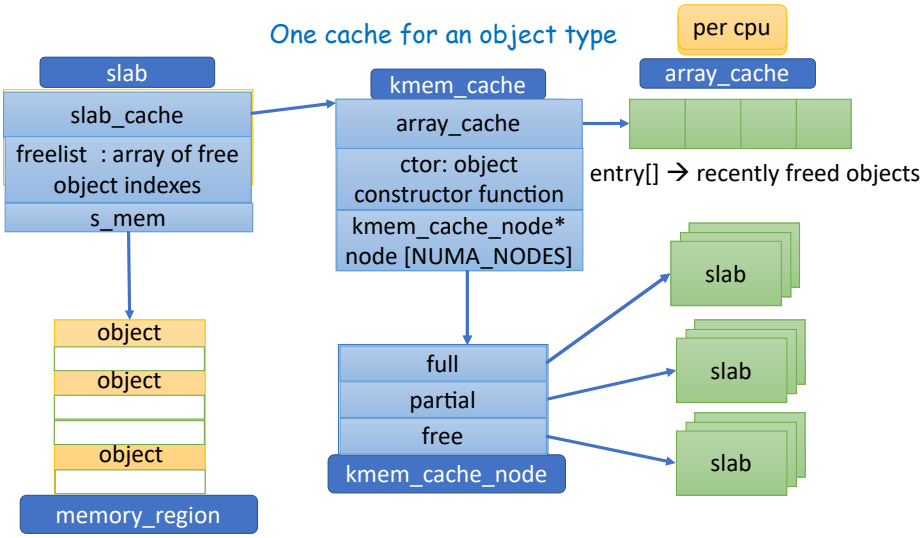


Figure 6.15: The slab allocator
mm/slab.h

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 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 as well as 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 even have a single allocated object. The algorithm is to first query the list of partially full slabs and 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. We add it to the `array_cache`. We also 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 6.16.

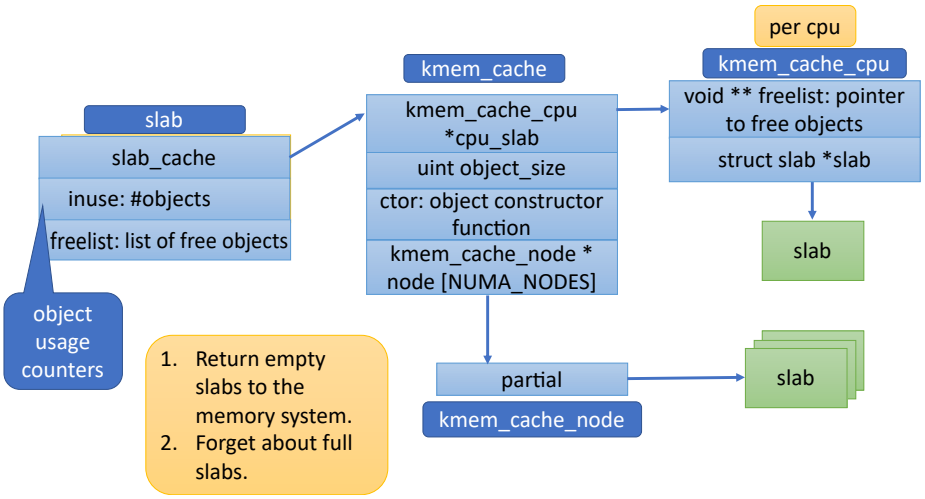


Figure 6.16: The slub allocator

We reuse the same slab that was used for designing 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 part in Figure 6.16 and just summarized it. This is because it has been shown in its full glory in Figure 6.15.

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 fully free and full slabs. We just forget about them. We only maintain a list of partially full slabs. We 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. Recall that the slub allocator heavily relies on pointer arithmetic.

Note that we do not maintain a list of full slabs nor empty slabs. Instead we chose to just forget about them. Now 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 a 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 back 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. An empty slab can be obtained from the buddy allocator, and it can be used for object allocation. This will make it partially full. It will be added to the slab cache. This keeps things nice, fast and simple – we maintain much less state.

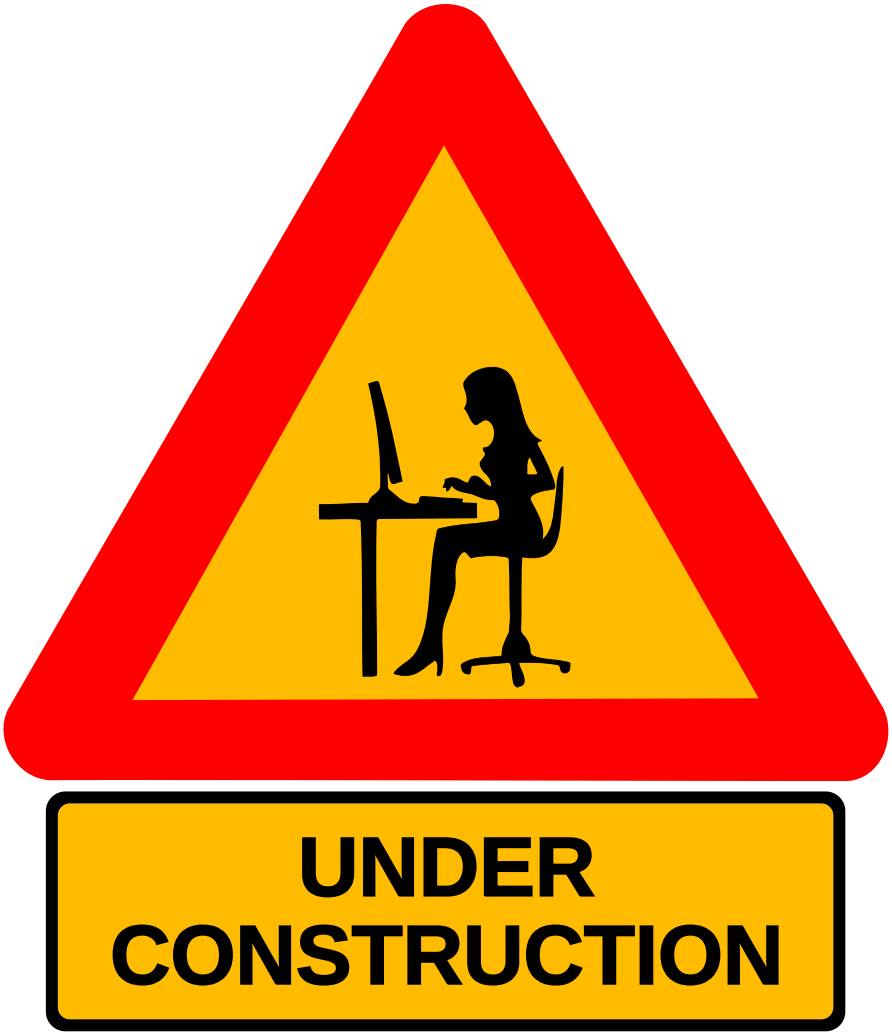
Chapter 7

The I/O System and Device Drivers


UNDER
CONSTRUCTION

Chapter 8

Virtualization and Security



Appendix A

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 book 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 starting from 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

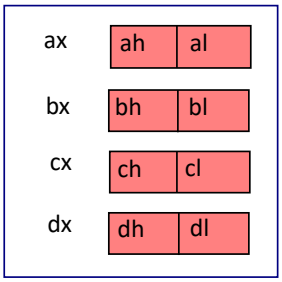


Figure A.1: Two virtual 8-bit registers within a 16-bit register

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. 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 and so forth. Furthermore, in the 64-bit variant the prefix 'e' was replaced with the prefix 'r'. Along with this 8 new registers were added from `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).

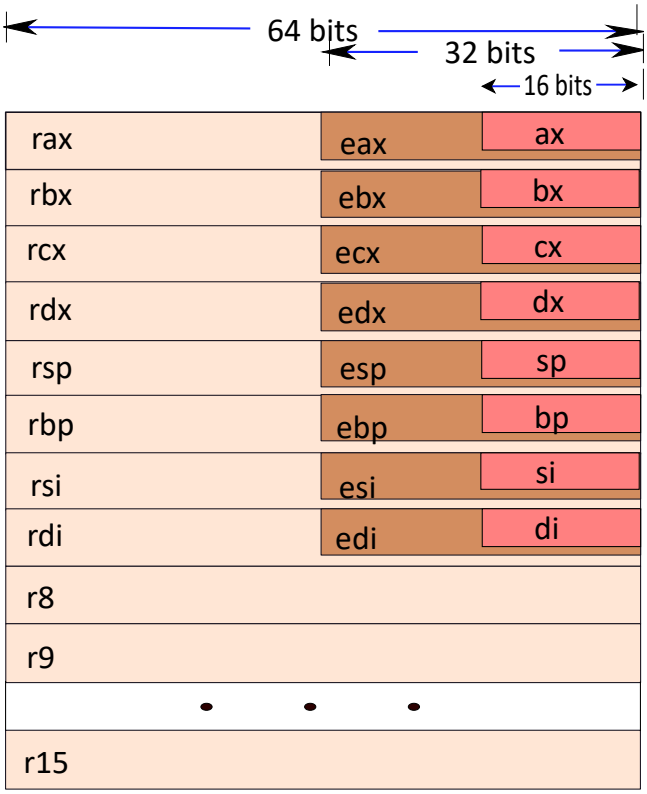


Figure A.2: The registers in the x86-64 ISA

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 compare instructions. Subsequent branch instructions

use the result of the last compare instruction for deciding the outcome of a conditional branch instruction. Refer to Figure A.3.

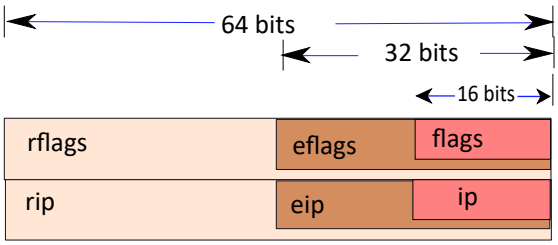


Figure A.3: Special registers in the x86 ISA

There are a couple of flag fields in the `rflags` register that are commonly used. These are bit positions. 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 bit 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 that 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`. Every register is basically a position in this model. It is true that registers are also directly accessible. For example, we can specify a value using 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. Sadly, almost all floating point operations are structured as operations that use the top of the stack (`st0`) as the implicit operand, and a push or pop is involved. 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.

Figure A.4: Floating point registers

The stack-based model is typically adopted by very simple machines that should be easy to program. We basically restrict our set of instructions to arithmetic instructions, load/store instructions, push and pop. The 80387 math co-processor that used to be attached to erstwhile Intel processors to provide floating point capabilities used this stack-based model. This basic programming model has continued to remain in the x86 ISA. Because backward compatibility is a necessary requirement, this model was retained to this day. 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 given additional arguments specifying that assembly code in the Intel format needs to be generated. 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, 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)` to 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. It is basically not possible to have two memory operands in an instruction.

Let us look at the code for computing the factorial 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 will be used for translating memory addresses. It can, for instance, be the code segment register.

$$address = base + index * scale + disp \tag{A.1}$$

`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 be scaled by a 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 from the start of the region. The `disp` field also can 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 are 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. The parse tree 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 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. 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 a function. 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 a time, 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 hundreds 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. The moment we have many files, the addresses of many variables and functions will simply not be known at the compilation stage. Such matters have to be resolved later.

Let us now take a look at Figure B.1. It shows many phases of the 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 only for a given C file. It contains machine code corresponding to the 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*. The linking process or the *linker* is then tasked with taking all the .o files and combining them into one large binary file, which can be executed by the user. This binary file has all the symbols' (let us assume this for the time being) addresses defined. Note that we shall refer to the final executable as the *program binary* or simply as the *executable*.

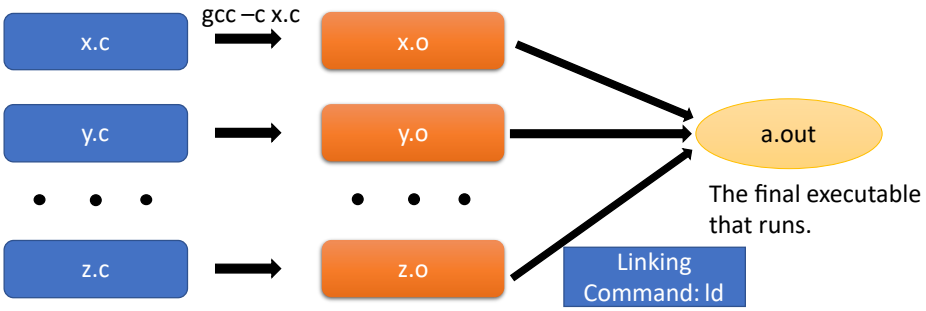


Figure B.1: The process of compiling and linking

It turns out that there is another serious problem. Whenever, a function is used in a C file, we need to know the *signature* of the function – 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 us, then the compiler 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 type conversion hierarchy is used. But to do that we need to insert 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 C program seamlessly. To summarize, with the signature we can solve many problems like automatic type conversion, correctly arranging the arguments and properly typecasting the return value. During compilation, the address of the function may not be known if it is defined in another C file. This is something that the linker will need to resolve as we have discussed.

Let us further delve into the problem of specifying function signatures, which will ensure that we can at least compile a single C program correctly and create object files, which the linker can process.

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. Often the keyword *extern* is used in C and C++ programs to indicate that the function is actually defined somewhere else.

Here the *definition* basically refers to the function's 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 a 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 correctly.

We need to do something similar for functions defined in other files in a large multi-file 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 literally copy-pasted into source code files.

A header file or a .h file precisely does this. It simply 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 could 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.

Barring a few exceptions, header files never contain function definitions or any other form of executable code. Their role is not to have regular C statements. This is the role of regular source code files and header files should be 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.

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 ensuring 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 factorial.h file. It is being included by specifying its name in between double quotes. This basically 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' 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 can now be used by other C files whenever they want to use the factorial function. They know that it is defined there. The signature is always available in factorial.h.

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 was enough information for the compiler to compile prog.c.

The other interesting part of having a header file is that the linkage between the signature and the implementation is somewhat dehyphenated. 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 The 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 files. Along with that any C file uses 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 link these library object files as well. In this case, we think of the standard library as a library of functions that make it easy for accessing system services such as reading and writing to files or the terminal.

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.

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.

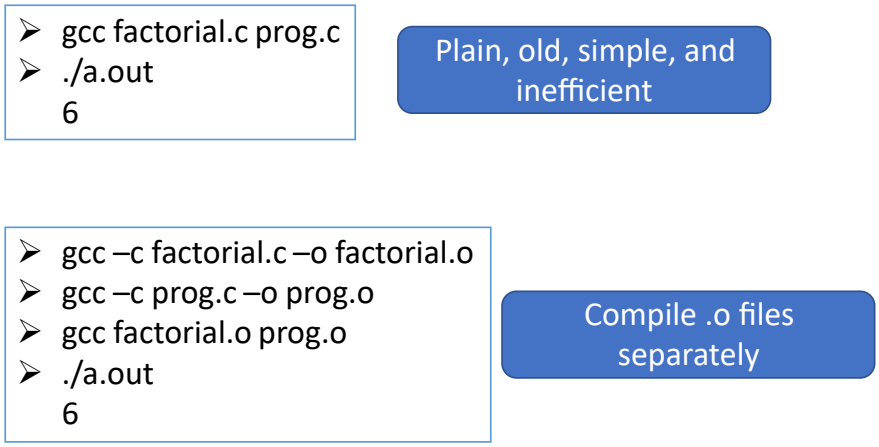


Figure B.2: Code for static linking

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.

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 files 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.

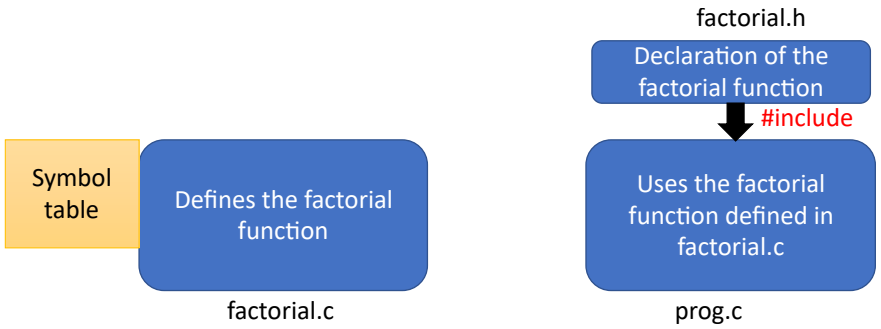


Figure B.3: Compiling the code in the factorial program and linking the components

Pass 1: It scans through all the object files and concatenates all the text sections (instructions), global variable, function definition 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 (uninitialized global/static variables) and constants. All the program code and symbols can be concatenated and the final addresses of all 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.

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 as a part of the first pass.

Issues with Static Linking

Let us now look at some of the 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 in the compilation options. We can check this using the `ldd` command. The output shows that the executable is statically linked. 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). 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 has 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 should not be made. Hence, we conservatively assume that every single library call that is made in any object file will actually be made and it is not dead code. Consequently, the code for all those library functions (and 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. Because of this, we need to include a lot of code in executables and hence 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 use some of the library functions defined in the standard C library, even if they are written in a different language. 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 in this process. 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.

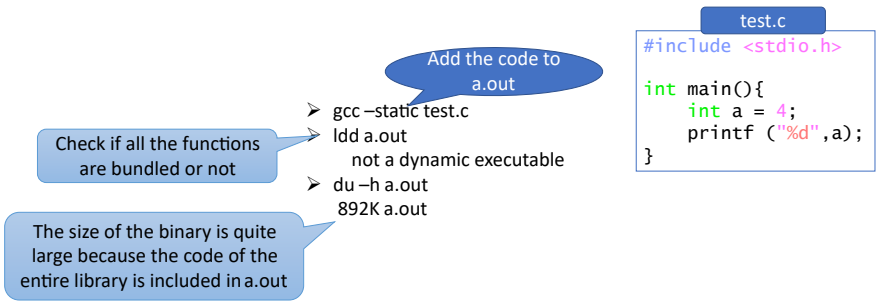


Figure B.4: Size of a statically linked executable

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.

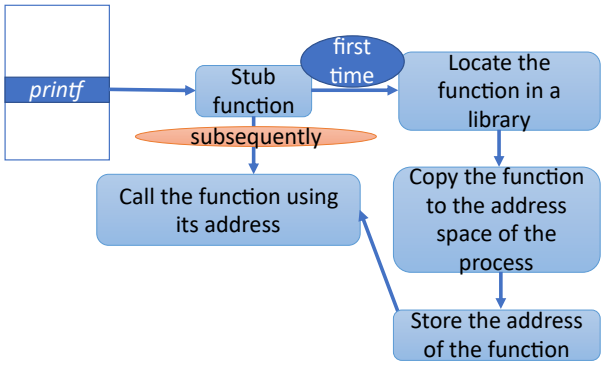


Figure B.5: Dynamically linking a program

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 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 run on the same machine, at runtime the stub function will call a function called `dlopen`. When invoked for the first time for a given library function, its job is to locate the library. Similar 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 other packages that provide some shared libraries that this executable requires need to be present in the system. Hence, it is essential that they are installed first. Further, they 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 popularly 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.

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. The archive or `ar` command then creates a library out of an object file. The extension of the archive is '.a'. It is not a shared object/library yet. It is a static library that can only be statically linked.

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 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 using `gcc`. This generates `libfactorial.so`. Next, we need to compile and link `prog.c` with the dynamic library that we just

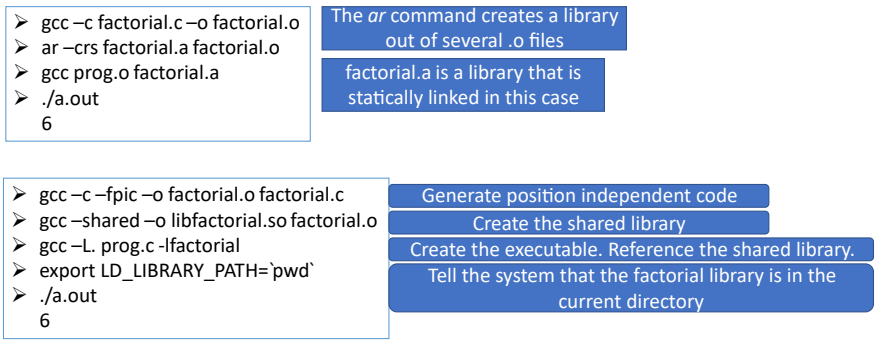


Figure B.6: Code for dynamic linking

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 `factorial.so` in the current directory because of the -L and -l flags.

However, 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 safely 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 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 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 specifications about the ISA and machine compatibility. It is divided into a set of 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). That 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 file), 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 with regards to dynamic linking in the dynamic section.

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). 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 dynamic section in the process's memory image. It 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 operating systems. 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 know when and where a data structure should be used and what it is good for. Furthermore, the reader should also 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. This is because if we want to write routines that operate on linked lists, then they will have to take a generic `void *` pointer as an argument for the encapsulating object (structure), and somehow find a handle to the `next` member. This is very tough and is also not a generic solution. 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.

Linux's solution is quite ingenious. It heavily relies on C macros. We would advise the reader to go through this topic before proceeding forward. Macros are very useful yet very tricky to understand.

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` namely `next` and `prev`. This is all that there is to the definition of a linked list. As we see `struct list_head` is in some sense representing a linked list node; it has pointers to

the next and previous entries. This is enough information to operate on the linked list. For example, we can add new entries as well as remove entries. The crucial question that we need to answer here is, "Where is the encapsulating object that needs to be linked together?" In general, we define a linked list in the context of an object. We interpret the linked list to be a list of objects. Here, we do not see an object with its fields, instead we just see a generic linked list node with pointers to the next and previous nodes. 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

```
struct list_head {
    struct list_head *next, *prev;
}
```

We will use two macros to answer this question as shown in Listing C.2.

Listing C.2: The `list_entry` and `container_of` macros
source : include/linux/list.h and
source : include/linux/container_of.h (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))); })
```

Let us start with describing 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. Consider the structures in Listing C.3. In the case of `struct abc`, the value of `offsetof(list,abc)` is 4. This is because, we are assuming the size of an integer is four bytes. This integer 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 as well as float is 4 bytes each. Hence, `__mptr` − `offsetof(type,member)` provides the starting address of the structure that can be thought of to be 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.

Listing C.3: Examples of structures

```
    struct abc {
        int x;
        struct list_head list;
    }
    struct def {
        int x;
```

```
        float y;
        struct list_head list;
    }
```

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 subtraction operation.

It is important to understand that a macro is a special function that is run by the preprocessor. However, it is quite different from a regular function. Its job is to generate code not a result as a typical function. 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 is 8. At runtime, it is quite easy to compute this given a pointer (`ptr`) to a `struct list_head`. The `list_entry` macro is simply a synonym of `container_of`. Their signatures are identical.

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 current node that we are considering is called `current`. To find the next node (the next one after `current`), which will again be 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 thus `struct abc` and the member is `list`. The `list_entry` macro will internally call `offsetof`, which will return an integer. This integer will be subtracted from the starting address of the `struct list_head` member in the next node. This will provide the pointer to the encapsulating object.

This is a very fast and generic mechanism to traverse linked lists in Linux that is independent of the type of the encapsulating object. Note that we can stretch this discussion to create a linked list that has different kinds of encapsulating objects. Note that, in theory, this is possible as long as we know the type of the encapsulating object for each `struct list_head` on the list.

C.1.1 Singly-Linked Lists

Listing C.5: Example of code that uses the `list_entry` macro

```
struct hlist_head {
    struct hlist_node *first;
};

struct hlist_node {
```

```
    struct hlist_node *next, **pprev;
};
```

Let us now describe singly-linked lists that have a fair amount of value in kernel code. Here the explicit aim is a one-way traversal of the linked list. An example would be a hashtable where we resolve collisions by chaining entries that hash to the same entry. Linux uses the `struct hlist_head` structure that is shown in Listing C.5. It points to a node that is represented using `struct hlist_node`.

It is a very simple data structure. It has a `next` pointer to another `hlist_node`. However, note that 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. This is where a small optimization is possible and a few instructions can be saved. We actually store a pointer to the next member in the previous node of the linked list in the field `pprev`. The advantage of this is that we can directly set it to another value while deleting the current node. We cannot do anything else easily, which is the intention here.

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 a 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 great 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 a node that is the root of a subtree to any leaf node has the same *black depth*. Here, the *black depth* of a leaf node is defined as the number of black nodes that we cross while traversing from the root of the subtree to the leaf node. In this case, we are including both the root as well as the leaf node.
5. If a node has exactly one non-leaf child, then its 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. Specifically, the property that this tree guarantees is as follows.

> The maximum depth of any leaf is at most twice the minimum depth of a leaf.

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.

They require 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, which is self-balancing. In this case, a node can have more than two children; quite unlike a red-black tree. This is a balanced tree and all of its operations are realizable in logarithmic time. 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, database and file systems tend to use B-trees quite extensively.

Let us start with the main properties of a B-tree of order m.

1. Every node has at the 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 k children, then it stores $k-1$ keys. These $k-1$ keys partition the space of keys into k non-overlapping regions. Each child then stores keys in the key space assigned to it. In this sense, an internal node's key acts as a key space separator.
5. All the leaves are the same level.

C.3.1 The Search Operation

Figure C.1 shows an example of a B-tree of order 3. "Order 3" means that every internal node needs to contain at least two children or alternatively at least one key. In the case of this example, the root stores two keys. 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 and the storage of values is not central to the operation of a B-tree.

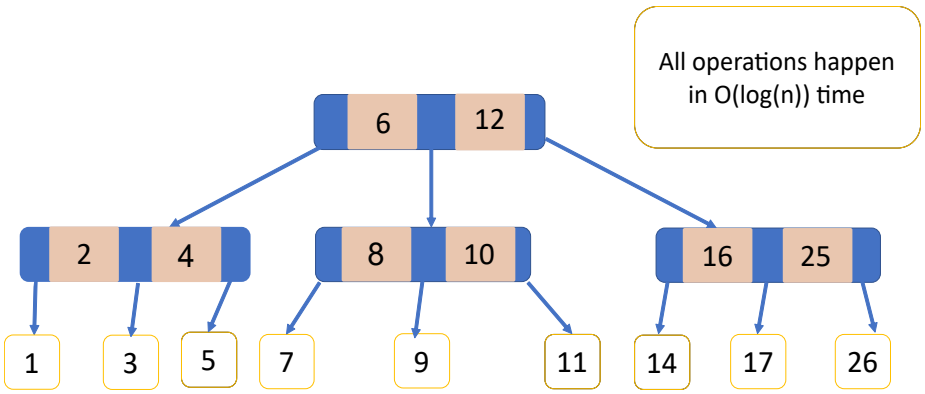


Figure C.1: Example of a B-tree

Now, if we consider the root node, we find that it stores two keys: 6 and 12. The leftmost child is an internal node, which stores two keys: 2 and 4. They point to leaf nodes that store a single key each. Note that as per our definition (order=3), this is allowed. Now, let us consider the second child of the root node. It needs to store keys that are strictly greater than 6 and strictly less than 12. We again see a similar structure with an internal node that stores two keys – 8 and 10 – and has three leaf nodes. Finally, the last child of the root only stores keys that are greater than 12.

It is quite obvious that traversing a B-tree is similar to traversing a regular BST (binary search tree). It takes $O(log(n))$ time. For added precision, we can account for the time that it takes to find the pointer to the right subtree within an internal node. Given the fact that we would have at most m subtrees, this will take $O(log(m))$ time (use binary search).

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. Note that if any new child is created, then a new key needs to be added to the parent node.

We need to understand that all these operations are not changing 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.

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.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 classical B-tree with additional restrictions. It 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.

Linux uses a Maple tree whose nodes are aligned to cache line boundaries. Furthermore, it is possible to process the tree in parallel and multiple users can seamlessly operate on different parts of the Maple tree. Furthermore, each key can either be a single value or can be a range, as is the case for VM regions (*start* and *end* addresses).

C.5 Radix Tree

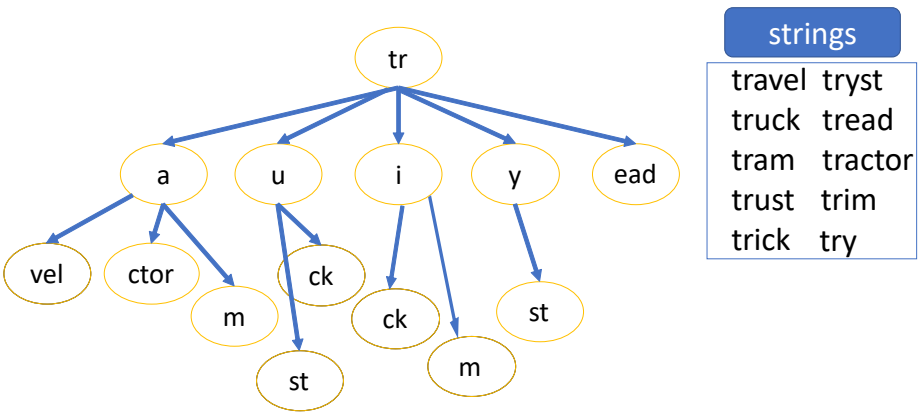


Figure C.2: Example of a Radix tree

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 as well as the internal nodes.

The algorithm works on the basis of common prefixes. 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 in a similar fashion 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 will be 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 in 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 Van Emde Boas Tree

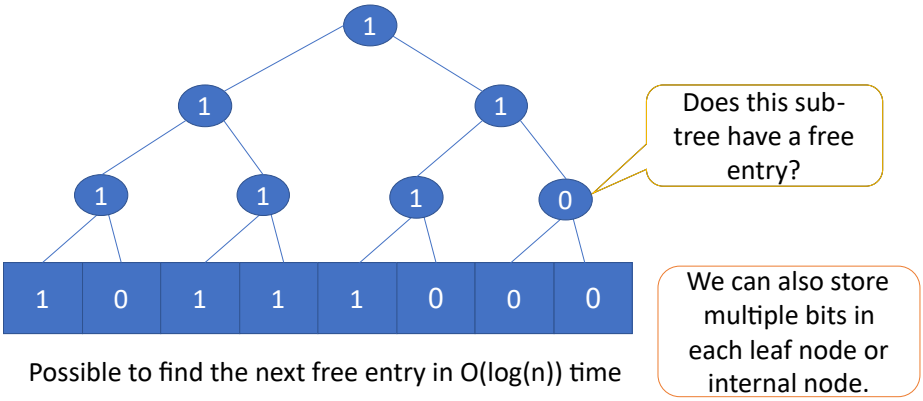


Figure C.3: Example of a van Emde Boas tree

This kind of a 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 i^{th} 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. On the same lines, we can have an analogous problem where the task is to find the first 1 in the bit vector. Regardless of whether we are searching for a 0 or 1, we need a data structure to locate such bits efficiently.

A naive algorithm is to of course start traversing the bit vector from the lowest address onwards 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 much. We clearly need a much faster algorithm, especially something that runs in $O(log(n))$ time.

This is where a van Emde Boas tree (vEB 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 vEB 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 vEB tree lies in the contents of the internal nodes. To explain this, let us start with the root.

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 1 value. Each of its children is the root of a sub-tree (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 underlying 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 second child. This process continues recursively until we reach the leaf nodes. We are ultimately guaranteed to find either a 1 or conclude that there is no entry in the bit vector that is a 1.

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 and the process will continue towards the root.

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.

[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.

[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`.

[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.

[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

__restore_rt, 117

, 16

Abstract Syntax Tree, 241
Anonymous Memory Region, 59
APIC
    I/O APIC, 88
    LAPIC, 88
arch_spinlock_t, 156
ASID, 212
Atomic Instructions, 123
Atomicity, 134

B+ Tree, 261
B-Tree, 259
Banker's Algorithm, 175
Barriers, 145
Base-Limit Scheme, 191
Belady's Anomaly, 199
Best Fit, 192
Bottom Half, 78, 97
Buddy Allocator, 222
Busy Waiting, 123

CAS, 133
cgroup, 183
Chipset, 41
Circular Wait, 126
Compare and Swap, 133
Compatibility Problem, 26
Compiler, 241
Compiler Pass, 242
Concurrent Algorithms, 134
    lock free, 140
    obstruction free, 140
    progress guarantees, 139
    theory, 134
    wait free, 140
Condition Variables, 143
Containers, 61
Context, 21
Context Switch, 75
    interrupt, 77
    process, 76
    thread, 77
    types, 76
Copy on Write, 70
copy_process, 72
Core, 16
CPL Bit, 18
CR3, 205
Critical Section, 121
Current Privilege Level, *see also* CPL Bit

Daemons, 102
Data Race, 124
    concurrent access, 124
    conflicting access, 124
Data Races, 119
Data Structures, 255
Dead Code, 242
Deadline Scheduler, 190
Deadlock Avoidance, 129
Deadlock Conditions, 126
Deadlock Prevention, 128
Deadlock Recovery, 129
Deadlocks, 126
Dining Philosopher's Problem, 127
Direct Memory Access, 44
DLL, 251
DMA, 44
Dynamic Linking, 250
Dynamically Linked Library, 251

Earliest Deadline First Algorithm, 168
EDF Algorithm, 168
ELF, 252
enum zone_type, 217
Exception, 19
Exception Handling, 98
Exec System Calls, 73

External Fragmentation, 192

Fence, 138
Fence Instruction, 124
Fetch and Increment, 132
FIFO Page Replacement, 199
FIFO Scheduling, 170
File-backed Memory Region, 59
First Fit, 192
Folios, 209
Fork System Call, 68
Frame, 33
Futex, 150

Happens-before Relationship, 124
Hardware Context, 75
Header File, 244
Hold and Wait, 126
Huge Pages, 204

I/O APIC, 89
I/O Ports, 42
I/O System, 40
IDR Tree, 64
IDT, 83
IDT Table, 94
idt_table, 94
Inter-processor Interrupt, 24, *see* IPI
Internal Fragmentation, 29, 192
Interrupt, 19
Interrupt Context, 97
Interrupt Descriptor Table, 83
Interrupt Handler, 78
IPI, 89
iret, 80
IRQ, 88
IRQ Domain, 93
irq_handler_t, 104

Jiffy, 24
Jump Table, 250

Kernel Memory Allocation, 222
Kernel Mutex, 160
Kernel Panic, 99
Kernel Stack, 53, 54
Kernel Threads, 74
kmem_cache, 62
KSW Model, 167

LAPIC, 89
Lazy TLB Mode, 213
Legal Sequential Execution, 135
Linearizability, 135, 136
Linker, 243, 246
Linux
    memory management, 58
    versions, 11
List Scheduling, 173
Loader, 253
Lock, 121
Lock Inversion, 163
Lock-free Algorithm, 132
Lock-Free Algorithms, 140
Lockdep Mechanism, 162
Lost Wakeup Problem, 143
LRU Algorithm, 196

Makespan, 165
Maple Tree, 261
Memory Barrier, 124, 138
Memory Consistency, 137
Memory Model, 137
Memory-Mapped I/O, 43
Monitor Lock, 183
Motherboard, 41
Multi-Threaded Process, 48
Multicore, 15
Multicore Scheduling, 172
Mutex, 150
Mutual Exclusion, 127

Namespaces, 61
Next Fit, 192
Nice Value, 56
No Preemption, 127
Non-blocking Algorithm, 132
Non-Blocking Algorithms, 134
Northbridge Chip, 41
NP, 173
NP-complete, 173
NUMA
    node, 219
NUMA Machine, 214
NVM Devices, 216

Obstruction-Free Algorithms, 140
Optimal Page Replacement, 196
Overlap Problem, 26

Page, 33

Page Table, 33, 205
Page Table Entry, 206
Patricia Trie, 262
PCID, 212
Per-CPU Region, 54
Phasers, 145
Physical Address, 30
Physical Memory, 214
pid, 60
    allocation, 64
Pool, 62
Preemption, 51
Process, 48
    creation, 67
    destruction, 67
Process Descriptor, 48
Process Id, *see* pid
Program Order, 137
Programmable Interrupt Controller, *see* APIC
Properly-Labeled Programs, 125
Pthreads, 130
PTrace, 67

Queues, 147

Radix Tree, 261
RCU, 164
Reader-Writer Lock, 144
Reader-writer Lock, 153
Real-time Scheduler, 190
Real-Time Task, 55
Red-Black Tree, 258
Registers, 16
    general purpose, 17
    privileged, 17
Relaxed Consistency, 139
Relocation Table, 247
runqueue, 183

schedule function, 180
Scheduling, 165
Scheduling Classes, 181
Sections, 217
Segmentation
    x86, 39
segmentation, 38
Semaphore, 151
Semaphores, 142
Sequential Consistency, 137
Shared Library, 251
Shortest Job First Algorithm, 167
Shortest Remaining Time First Algorithm, 169
Signal, 19
Signal Delivery, 109
Signal Handler, 20
Signal Handlers, 108
signalfd Mechanism, 114
sigreturn, 117
SIGSTOP, 52
Single Core Scheduling, 167
Single-Threaded Process, 48
Size Problem, 31
Slab Allocator, 227
Slub Allocator, 229
SMP, 154
Soft Page Fault, 199
Softirq, 101
Softirq Interrupt Context, 103
Software Context, 76
Southbridge Chip, 41
Spin lock, 123
SRTF Algorithm, 169
Stack Distance, 193
Stack-based Algorithms, 195
Static Linking, 247
struct free_area, 224
struct hlist_head, 257
struct idr, 62
struct irq_desc, 93
struct irqaction, 103
struct k_sigaction, 114
struct list_head, 256
struct mm_struct, 58
struct mutex, 160
struct page, 209
struct pglist_data, 219
struct pid, 61, 63
struct pid_namespace, 62
struct pool_workqueue, 107
struct raw_spinlock, 156
struct rq, 184
struct rt_sigframe, 116
struct sched_class, 182
struct sched_entity, 184, 185
struct sched_info, 57
struct sigaction, 115
struct sighand_struct, 114
struct signal_struct, 113

struct sigpending, 115
struct sigqueue, 115
struct sigset_t, 113
struct thread_info, 48
struct ucontext, 116
struct upid, 63
struct worker_pool, 106
struct zone, 218
Swap Space, 37
Symbol Table, 247
Symmetric Multiprocessor, 154
Synchronization, 119
sysret, 80
System Call, 19
System Calls, 84

Task, 48
Task Priorities, 55
Task States, 51
Test-and-set, 123
Test-and-set Instruction, 123
Thrashing, 202
Thread, 48
Thread Local Storage, 77
thread_info, 49
Threaded IRQ, 101
Threaded IRQs, 103
Timer Interrupts, 22
TLB, 36, 211
Top Half, 78, 97
Translation Lookaside Buffer, *see* TLB
TTAS Lock, 122
Two-Phase Locking, 128

Unlock, 121

Van Emde Boas Tree, 263
vector_irq, 96
Virtual Address, 30
Virtual Memory, 24, 32
vm_area_struct, 59
vruntime, 186

Wait-Free Algorithms, 140
Wait-Free Queue, 148
Weak Memory Models, 138
Work Queue, 101
Work Queues, 104
work_struct, 105
Working Set, 201
workqueue_struct, 105
Worst Fit, 192
Wrapper Class, 107
WS-Clock Algorithm, 197
WS-Clock Second Chance Algorithm, 198

x86 Assembly, 235
    floating point instructions, 237
    instructions, 238
    memory operands, 239
    registers, 235

Zombie Task, 52
Zones
    sections, 211