OPERATING SYSTEMS

A MULTI-PERSPECTIVE EPISODIC APPROACH

1ST EDITION

Jae Oh

OPERATING SYSTEMS

A Multi-perspective Episodic Approach

First Edition

By Jae Oh
Syracuse University

cognella® | ACADEMIC PUBLISHING

Bassim Hamadeh, CEO and Publisher
Kassie Graves, Director of Acquisitions
Jamie Giganti, Senior Managing Editor
Miguel Macias, Graphic Designer
Mieka Portier, Acquisitions Editor
Sean Adams, Project Editor
Luiz Ferreira, Licensing Specialist

Printed in the United States of America

ISBN: 978-1-5165-0701-6 (pbk) / 978-1-5165-0702-3 (br)

Contents

Preface

An operating system (OS) is one of the most comprehensive computer software systems we encounter every day. Understanding operating systems is challenging for undergraduate students, who most likely have only a couple of years of computer science study. This book is written to guide students without much maturity in computer science to understand operating systems.

However, this book is aimed for undergraduate students who have taken fundamental computer science courses, such as data structures and minimum understanding in computer organizations. This book will assume only the knowledge of basic data structures and therefore is suitable for sophomore or junior level of undergraduate students.

There are many textbooks on operating systems. Why write another OS textbook? In my opinion, some existing textbooks are theoretical and written largely as reference books for OS concepts. Some books are practical, presenting many code examples; however, the materials are challenging for undergraduate students who do not have a certain maturity level in computer science. Reading and understanding these books require readers a lot of time. In the Curriculum Guidelines for Undergraduate Degree Programs in Computer Science (CSC2013)—a report by the Joint Task Force on Computing Curricula composed of delegates from the Association for Computing Machinery (ACM) and the Institute of Electrical and Electronics Engineers (IEEE) Computer Society—the amount of curriculum core hours for operating systems topics is reduced to fifteen hours in ACMs recommendations in 2008 and from eighteen core hours in 2001.

Some existing textbooks have more practical discussions that gives some implementation details; however, the materials presented are still not well accessible to undergraduates and the general public. I believe that one of the most important way of understanding an OS is being able to see the big picture, seeing many different computer users' perspectives simultaneously. An OS developer must be able to see how the average users would use the computer, how application programmers would develop programs, and how

system administrators would administer the computer. An OS developer should also understand how other OS developers use and extend the system.

There are wonderful ideas and algorithms that are used to build operating systems. Some of these algorithms are also quite useful for other areas in computing. Most of existing OS books do not explain how these algorithms are also relevant to other computing areas. This book will attempt to do that as well.

I started to envision this book as the result of teaching undergraduate and first-year graduate operating systems courses for the last fifteen years as an instructor. Although I started out using a well-known textbook in the beginning, fourteen years after, my lecture materials have evolved into a very different set of lectures.

One of the approaches taken in this textbook is to start with the construction of very simple OS mechanisms then discuss the limitations of the system to introduce remedies for the limitations. The discussion section of each unit of learning will introduce a way to improve the current system. Unlike most other OS books, I will try not to present various components of OS in a disconnected way. Rather, I will start explaining a very simple OS for a simple architecture, then gradually discuss more complex systems. This approach will inevitably bring out discussions on various components of an OS in every part of this book. My goal is to give students a whole picture of the entire system each time, whether it is a simple or a complex system.

The most important learning outcome of this book is not about building operating systems but about engaging students in *system-oriented thinking*.[1] In other words, I want to enable students to start reading existing OS source code, make sense out of it, and be able connect all the major levels of abstractions in the computer system–the machine level, the OS level, the system program level, the application program level, and the end user level.

This book is written in a multi-perspective episodic approach. It attempts to presents concepts from multiple perspectives: the end users, the application programmers, the system programmers, the OS designers and developers, and the hardware engineers. It is written in an episodic way, providing examples of interactions of various components for certain scenarios.

[1] Alluding to *computational thinking*

1 Introduction

"One of the main characteristics of a computer science mentality is the ability to jump very quickly between levels of abstraction, between a low level and a high level, almost unconsciously." – Don Knuth.

1.1 How to understand OS

One of the beginner's most challenging part of learning Operating Systems is assembling various concepts together in one coherent view. Operating Systems are "living and breathing" machinery, behaving in certain ways depending on situations. It is my belief that an operating system book should be studied in a "what if" fashion.

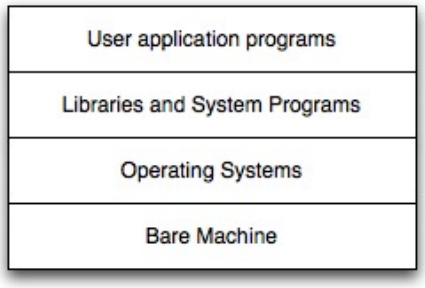


Figure 1.1: Computer system hierarchy

As in Figure 1.1, the operating system is situated in between the hardware and application programs. An OS utilizes hardware and software resources and abstracts away details of hardware services for various users. It attempts to maximize the usages of a computer's hardware components such as the Central Processing Unit (CPU), memory, and Input/Output (I/O) devices and software resources such as files. It also hides the messy details of underlining hardware from users. The "users" of an operating systems are in several categories: (1) the end user who uses the computer for web browsing, word processing, etc. (2) the application programmer who develops programs such as web browsers, word processors, and any programs that runs on an OS, and (3) the system administrators who manages the computer system, creating and managing user accounts, etc.

I would like to give one word of advice to readers of this book on how to study with this book (or any other OS books):

Advice: To understand operating systems, you must be able to understand all these users' perspectives, quite often simultaneously.

Therefore, in this book, we will often start explaining a concept from those users' perspectives and then what OS developers have to do to support the needs of these users. In fact, this is a very unique approach to the subject. Indeed many OS textbooks do implicitly assume this approach. However, for beginners, it is utmost important to align students to the correct perspective each time a new concept is explained. It is indeed my experience that students are often prevented from understanding simple OS concepts due to mis-aligning themselves to an incorrect perspective.

Therefore, in this book, we will often start explaining a concept from

those users' perspectives and then explain what OS developers have to do to support the needs of these users. In fact, this is a unique approach to the subject. Indeed, many OS textbooks do implicitly assume this approach. However, for beginners, it is utmostly important to align students to the correct perspective each time a new concept is explained. It is indeed my experience that students are often prevented from understanding simple OS concepts because these students have mis-alignined themselves to an incorrect perspective.

1.2 Why do we need an operating system?

Imagine that you have just bought a computer system that didn't come with an operating system. When you turn on this computer, what do you expect to see? The most likely message you will see on the screen is "no operating system found" or something similar. But do you know what had happened until the message was displayed? To answer this question, it would help to know what happens when we turn on a computer with an operating system installed in its hard disk.

Figure 1.2: A login screen of the Ubuntu Linux System

We call the process of bringing OS up and running from the time a computer is turned on as *booting*. When you press the ON switch, the following happens until the user sees a login screen (See Figure 1.2).

1.3 Understanding how the boot process works

The boot process can be dened as the time from the ON switch is pressed to the time the user sees a login screen. All computing devices have a boot process, during which the device goes through some initial congurations until the device is ready for user commands. Simpler devices may have a simpler and shorter boot process. An analogy is that an automobile has a simpler starting process than an airplane. Lets discuss a common boot process in computer systems.

1.3.1 What does the Central Processing Unit (CPU) do?

In order to understand the boot process, we need to first understand how the CPU works. Figure 1.3 shows a simplified CPU architecture. This is what you would "see" if you could open up a CPU chip. The figure shows an arithmetic logic unit (ALU), a set of general purpose registers, and a program counter register. Commercially available CPUs are much more complicated, but the CPU in the figure captures essential components. If you have taken a computer organization or computer architecture course, you may have seen this diagram.

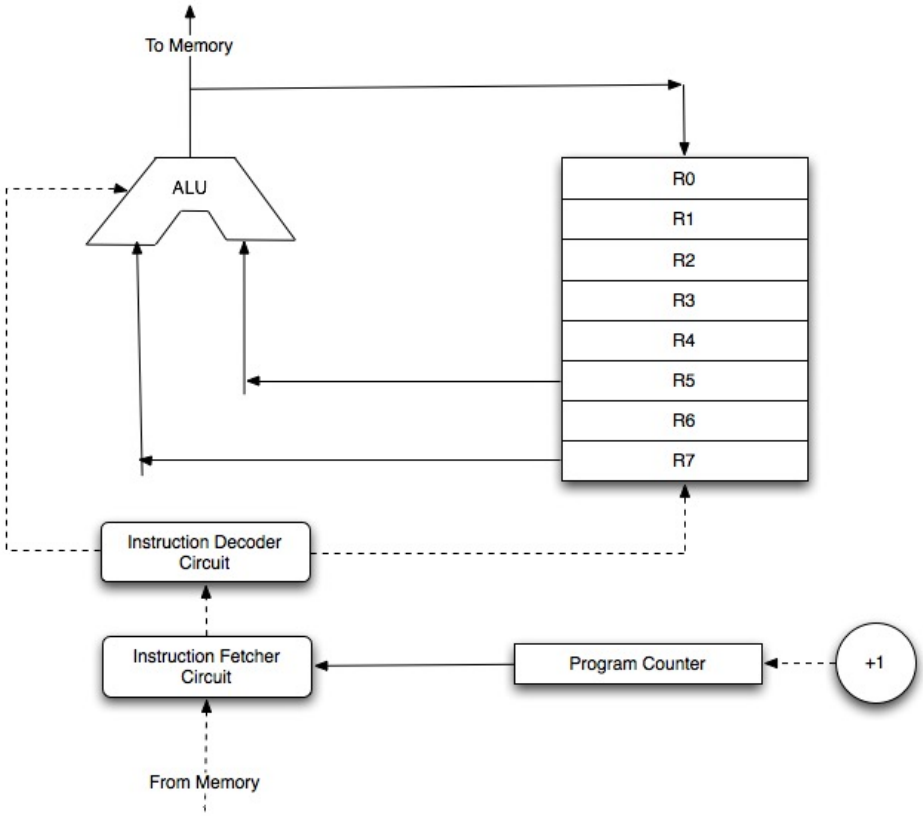


Figure 1.3: A simplified CPU architecture. At each execution cycle, the instruction fetcher circuit uses the program counter's value to fetch the instruction stored in the memory; the fetched instruction is decoded by the instruction decoder circuit, then it is executed by the ALU. The dotted lines are control signals and the solid lines represent data flows.

So what does the CPU do? We all know that it runs computer programs. But how? Think of the CPU as a machine. It is no different from any machines we use everyday. Consider a radio, for example. When you turn it on, it will receive the signals the tuner is adjusted to, transform the signals into audible sounds, amplify the sounds, and play the sounds. If you change the station by dialing the tuner knob, the radio will play a different station.

CPU is nothing different. When you turn on the CPU, it will try to find an instruction (or a command) to run.[1] Precisely speaking, the CPU's

[1] In a computer organization and architecture course, you learned about the instruction

logic circuit will check its program counter (PC) register value, in which the memory address of the next instruction to be executed is stored. Then the instruction in the memory is fetched to the CPU, decoded, and executed by the instruction fetcher circuit, the instruction decoder circuit, and the arithmetic logic unit (ALU) in the CPU, respectively.

As you may know, every program is translated to an equivalent binary program. A binary executable program is composed of a sequence of CPU instructions. Mathematically speaking, a CPU is a finite state automaton, as shown in Figure 1.4. A CPU will continue the fetch-decode-execute cycle as in the figure. The finite state machine shown in the figure only illustrates the core states of the CPU. For example, each instruction, such as "ADD,"[2] is also a finite state machine. Therefore, a real CPU can have over several thousands or more states depending on the complexity of the CPU.

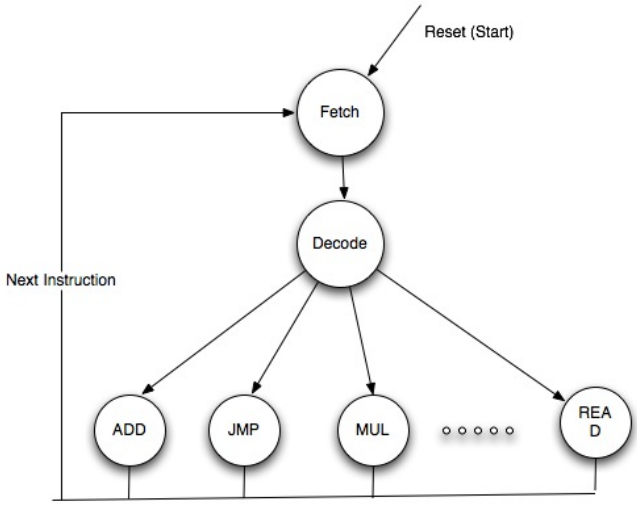


Figure 1.4: A CPU is just a finite state machine

set. Every CPU has a set of predetermined instructions it understands and is able to execute. Some common examples of the CPU instruction set are `ADD (for addition of numbers)`, `MUL (for multiplication of numbers)`, etc.

[2]ADD is a typical symbol for an addition instruction for a CPU

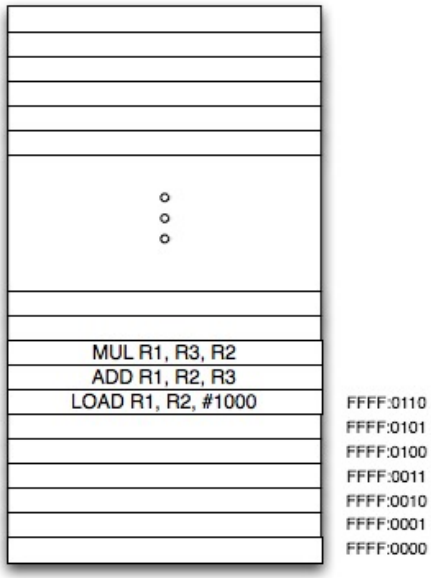


Figure 1.5: A program to be loaded. The program starts from address FFFF:0110.

Let's assume that the program in Figure 1.5 is stored in the memory address FFFF:0110. To have the CPU to run the program, all we need to do is to store the beginning address, in this case, FFFF:0110, of the program to the program counter register of the CPU. Then the CPU hardware logic is designed to look into the PC and fetch the instruction stored in the address, decode the instruction, and execute it.[3]

1.3.2 The boot process and operating system Initialization

Now that you understand how CPU executes an instruction from the fetch-to-execute cycle, it is time to discuss the entire boot process. When you first turn on a computer, what happens inside of the computer? In general, the first operation performed by the CPU is to fetch the instruction stored at a designated/pre-determined memory address. This address can be set by hardware. For example, an old CPU called the Z80 CPU immediately begins executing code from memory address 0000 when it is in the reset state.[4] If you want the CPU to run a program stored in a different location,

[3]Keep in mind mind as you read the boot process explained.

[4]A reset is done when the CPU gets turned on or a reset signal is sent to the CPU.

you can simply load the starting address of the program in the PC.[5] Think about a person who always does the same thing when she wakes up in the morning. CPU works the same way!

Let's follow, in step-by-step, what happens from the time when you turn on the computer and until you see an operating system waiting for your command.

1. **The user turns on the computer:** When the power is applied to the system and all output voltages from the power supply are good, the power supply generates a "power good" signal to the motherboard logic. In turn, a reset signal to the CPU is generated, causing the CPU to begin processing instructions. The very first instruction the CPU can execute is the JUMP instruction. The JUMP instruction causes the CPU to set the program counter value to the address of the next instruction to execute. By executing the JUMP instruction, the CPU's PC is set to the first instruction's address in the boot program, Basic Input/Output System (BIOS), which is stored in the program counter. In other words, the JUMP instruction will cause the CPU start executing the BIOS program. The first instruction's address is predetermined.

2. **Running the boot program:**

   The boot program performs various initializations and tests hard- ware components such as the motherboard, the CPU, the video controller, and peripheral controllers. The boot program is usually stored in a programmable read-only memory (PROM), which is called *firmware*. PROM is usually programmed using a PROM burner. Firmware is somewhat between hardware and software that is stored in a read- only memory. Once it is written with a burner, it is not usually changed, although it is still possible to rewrite PROM for firmware updates. As you boot a computer, you may remember seeing a message about video card, for example; and this is an indication of the video card being tested by the boot program.

3. **Looking for an Operating System:**

[5]Instead of having a fixed starting address, some CPUs have a so-called reset vector that contains the address of the first program to be executed. The location of the reset vector, which is usually the highest memory address, is known to the CPU. In x86 CPUs, the Intel's CPU family, the reset vector address is FFFF:0000h. In this address, a JUMP instruction with the first instruction's address of the boot program is stored.

Looking for an operating system: Once the testing is complete and no errors are found, the boot program will begin searching for an operating system in a designated location–usually in a hard disk. The master boot record (MBR), which is found in the first track of the main hard disk, contains information about which track and sector an operating system can be found in. If a bootable operating system is not found in the hard disk or any other peripheral storage devices, a message, such as "no systems found," is displayed on the screen and the boot process stops with a failure. When a bootable OS is found, the OS is being loaded in the main memory.

4. **Running operating system:** Once the operating system is loaded in the main memory, the CPU's program counter is initialized with the first instruction of the OS. As we discussed, consequently the CPU will start executing the OS.

5. **OS initializes further components and runs system processes** Once the operating system starts, the OS will further initialize necessary hardware components for a software-based configuration process. Then the OS will start necessary processes (programs) for the system. These are called *system processes*. This will include the scheduler process, memory management processes, server processes, etc. For example, a computer that serves as a web-server will start a web server program, which will listen to incoming web page requests and send the requested page over the http protocol. We will discuss various system processes in Chapter 2.

After all the above steps are completed, the computer is ready for users!

1.4 Programming in historical computers

The early computers, in 1940s and 50s, were built with vacuum tubes and plug boards, which were similar to an old telephone operators board. Programming was done through rearranging the plugs. See Figure 1.6. There was no way to write a program the way we program today. In other words, there was no programming languages as we know it. Today, we write programs using one of the high-level programming languages, such as C++, then we us a C++ compiler to generate a binary executable of the program. To execute the program, the binary program is loaded in the main memory by OS and executed. In 1940s, programming languages were yet to be invented. Therefore, by rearranging the wires, IBM402 could do different

computations. Notice that this is a form of "programming" even if it is not with a programming language as we know it.

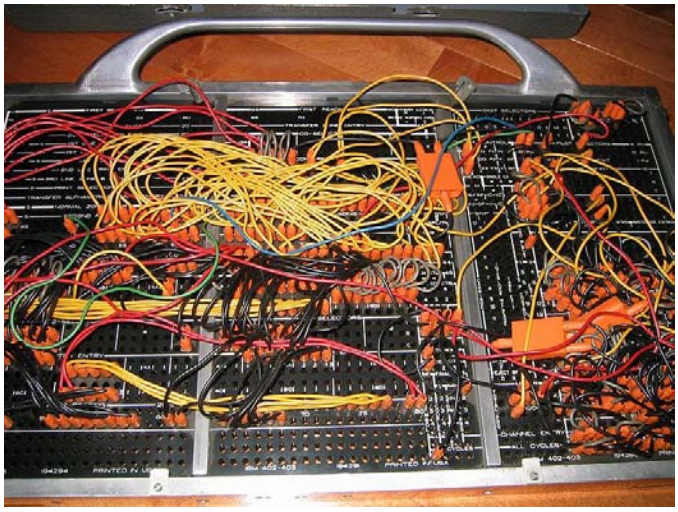


Figure 1.6: The IBM 402 Plug Board. This is a similar device to a telephone operator's switch board. By having a different wiring, a different computation can be done. This is a form of programming.

In 50s and 60s, transistors were invented, and computers had become accessible to more people. Still, these computers were only available to large organizations such as universities, companies, and government institutions, because they were very expensive. Programming, however, had gotten a bit easier in this period. In these systems, usually programming was done manually on paper. Then the programmer produced a deck of punch cards that represented the program using a card-puncher machine. Each card in the deck represented (more or less) one line in the program. See Figure 1.8. Then the deck was submitted to the computer operator. When a large enough number of decks were submitted by various users, the operator then took the decks of cards, each deck representing one program, and loaded them to a card reader machine. (See Figure 1.9). The card reader would read all programs in a big tape that was in turn loaded into the CPU machine. The CPU machine then executed each program one by one and wrote the output results to the output tape. The output tape was then loaded to the printer. The printer printed all output results one by one. The operator in turn separated the outputs for each user and placed the results in each user's mailbox.

Figure 1.7: An IBM card puncher machine. A card puncher machine is used to encode (more or less) each line of a program. A deck of card represents a program as shown in Figure 1.8.

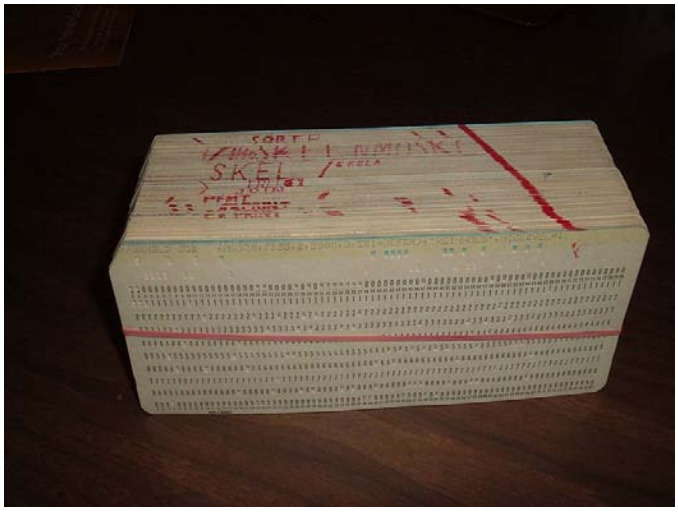


Figure 1.8: A deck of punch card representing a computer program written in the PL/1 programming language.

Figure 1.9: IBM 1442 punch card reader. Decks of cards are inserted to the card input, and the decks of cards are read into a tape storage.

Figure 1.10: IBM System 360 Tape Drives. The tape drive system usually consists of three tape drives usually: an input tape drive, the system tape drive, and the output tape drive.

Figure 1.11: A computer room with card readers, the main frame computer, tape drives, and a printer.

If there was an error in a users program, the output would reflect that error. The user then fixed the error in his/her program and recreated the card deck to submit. This whole process repeated until the programmer was satised with the result. As we can see, this was a very labor intensive and

time consuming process. Figure 1.11 shows such a computer room with all the machines described above.

In 1970s, many people started to show interests in personal computers. There were several efforts, but we will study Altair 8800 computer, which was introduced by the MITS corporation. Paul Allen and Bill Gates took this computer and wrote a Basic interpreter for the computer. Altair 8800 is a mostly bare-bone hardware computer ready to run programs. We will discuss how the Basic interpreter written for Altair 8800 enabled the computer for new potentials.

1.4.1 Single user and single process system

Consider a computer system that belongs to only one user, which means that it only allows one user to use the machine at any given time. It also can run only one program at a time. Old personal computers were used by one user and ran one program at a time. Altair 8800 is one of these computers. However, with enough memory, and a proper OS support, and time, the computer, although slow, is able to run multiple programs and serve multiple users concurrently.

Single User/Single Program Computers, Single User/Multi-Program Computers, Multi-User/Single Program Computers, and Multi-User/Multi-Program Computers

We define four different types of computers for this book.

- Single user/single program computers (SUSP): A SUSP computer can serve one user and run one program at a time. In this computer, "running one program at a time," means that once a user program is started, it will have to be explicitly terminated (was finished successfully, user decided to finish it, it crashed because of an error, etc) before we can run another program. Also, a SUSP computer is owned by one user. An example is an old game console such as Nintendo Game Boy.
- Single user/multi-program computers (SUMP): A SUMP computer is perhaps the computer we are most familiar with. These are personal computers of the present day, such as Windows or Macintosh computers. The computer is (most likely) owned by one user, but it is capable of running multiple programs *concurrently.* "Concurrently" means that multiple programs can be loaded to the memory at the same time, and the computer (in fact the OS) has the capability of switching from one program to another, back and forth, with or without the human user noticing the switchings. Think about how you use your computer now. You can switch from browsing web to word processing and back and forth. There are also switchings from one program to another without the user noticing it.

Single User/Single Program Computers, Single User/Multi-Program Computers, Multi-User/Single Program Computers, and Multi-User/Multi-Program Computers

- Multi-User/Single Program Computers: A particular example of this type is a batch processing computer. We discussed an old IBM computer system using card punchers, tape drives, etc. In the early days of batch processing computers, computers could only run one program at a time. This means that once the computer started a program, it would finish the program before starting a new one. Programs to be executed were queued up sequentially in the input tape drive and read one at a time to be executed. The output was written to the output tape drive one output at a time. When all programs were done, the output tape drive contained outputs of all the program that ran. Then the output tape was brought into the printer and the outputs printed out one by one in sequence. Each output would belong to a user's request. Therefore, the printout was given to the appropriate user. This type of system doesn't provide the capability to the users to interact with the computer during programs execution time.
- Multi-User/Multi-Program Computers: This type of computer is the so-called server machine. Let's talk about compute servers. Many universities provide computing services students and faculty can use. Users with a legitimate account can log in to the machine and use the machine for their needs. In this case, multiple users can log in to the system at the same time and multiple programs are executed. Each user can run multiple programs. A web-server machine is also such a machine. It listens to a network port (usually port 88) and serves incoming requests from various users from all of the world.

It is important to understand that depending on what type of computer we need to design, the operating system for the computer will have different capabilities, as well as different complexities.

Figure 1.12 shows the front panel display of an Altair8800 computer. By using flip switches, users can store programs one instruction at a time, then run the program stored. This is, as a matter of fact, programming. For the readers who are not sure about why this is programming, let's consider the program codes in Figure 1.13. The C program in Figure 1.13, initializes the variable `i` and `sum` to zero. Then the program increments the `sum` variable by 1 in each iteration of the loop; the loop is executed five times. We can envision an equivalent assembly code that does the same in the code in the right side of Figure 1.13, in which `$i` represents a memory location of the

variable `i` and the `LOAD` command loads zero to `i`. The same is done for the variable `$sum`. The symbol `$` simply represents that the following symbol is a variable name. The next command `ADDI` increments the value of the variable `$sum` by 1. The `BNE` instruction makes sure that the iteration is done 5 times. We all know that each line of the assembly code is directly mapped onto machine instruction represented by zeros and ones. Therefore, each of the five lines of the assembly code is directly corresponding sequences of zeros and ones. To run a program, we must load it to the computer's main memory.[6]

Altair8800 has flip switches that can represent an instruction. For example, `LOAD $i, 0` is represented by a binary code. Just for the sake of explanation, let's say that the code is represented by `1000 0000 0000 0001`, a 16-bit code. To load this instruction, the programmer would flip the sixteen switches on the front panel to `ON` for the first switch and for the last switch. All other switches are set to `OFF`. Then the programmer will press the `instruction load` switch on the panel. The programmer will repeat this process for the next instruction, i.e., `LOAD $sum, 0`. She will continue to do this until all program lines are loaded in the memory. To run the program, the `Run` switch can be pressed.

Figure 1.12: The Altair8800 computer's front panel

[6]ADDI is an instruction that adds the actual value, in this case 1, to `$sum`; BNE is an instruction that jumps to the target address, in this example, L1, if the first argument (`$i`) is not equal to 5.

```
int main() {                    main:
  int i = 0;                        LOAD $i, 0
  int sum = 0;                      LOAD $sum, 0
  for (i = 0; i < 5; i++)       L1: ADDI $sum, $sum, 1
      sum = sum + 1;                INC  $i
}                                   BNE  $i, #5, L1
```

Figure 1.13: C program and an equivalent assembly code

As anyone can agree, programming using flip switches in Altair 8800 is extremely cumbersome. We must realizes that this is not necessarily due to its hardware limit. Altair 8800 was designed for hobbyists who want to "play" with the hardware, write software for the machine and perhaps making it more useful or fun. Let's think about one way to make this computer a bit more easy to use. The goal is to make loading of programs easier. Remember, currently the program must use the flip switches to load a program. We can do an improvement to simplify this process. First, we write a loader program that will read a program from a peripheral device. We can utilize the idea of the punch card device. A loader program will read the program from the punch card reader and load the program in the memory. A loader program is relatively small so that it can be loaded in memory the usual way (using the switches). Once the loader program is loaded, any other programs can be loaded from a punch card reader connected to Altair 8800. With this loader, a basic interpreter can also be loaded in the memory. Once it is loaded and run, Altair 8800 becomes an interactive machine, running Basic. There are many other improvements we can make.

1.4.2 A simple computer architecture

Throughout the book, we consider a simple computer architecture described below. The hardware is simple, yet it can represent any computer that exists today. With a risk of over simplification, the advancement of computer hardware so far has been improving the speed and capacity. The fundamental computing paradigm has not changed at all; all computers are von-Neumann machines, which means programs are stored in memory and the CPU executes programs by fetching instructions from memory.

If you have taken a computer organization or architecture course, you may remember that a computer system consists of a CPU, memory, and I/O devices, at the very least. Altair 8800 is such a computer, for example.

We start with our idealized computer, which has these three components: a CPU, a memory unit, and an input device. Figure 1.14 shows this computer. In the figure, the CPU is connected to memory management unit (MMU). The MMU translates the memory access request to proper memory address and sends the request to the bus. Several controllers are connecting the I/O devices and the rest of the hardware. The figure shows a hard disk controller, a keyboard controller, a monitor controller, and a network controller. An I/O device must have a corresponding controller. Therefore, a keyboard device needs a keyboard controller that connects the keyboard to the bus. The same is true for other devices. We will discuss the details of interactions between these hardware components later. But for now, assume that when the load button is pressed, the program stored in a hard disk is loaded by the computer from memory address 0.

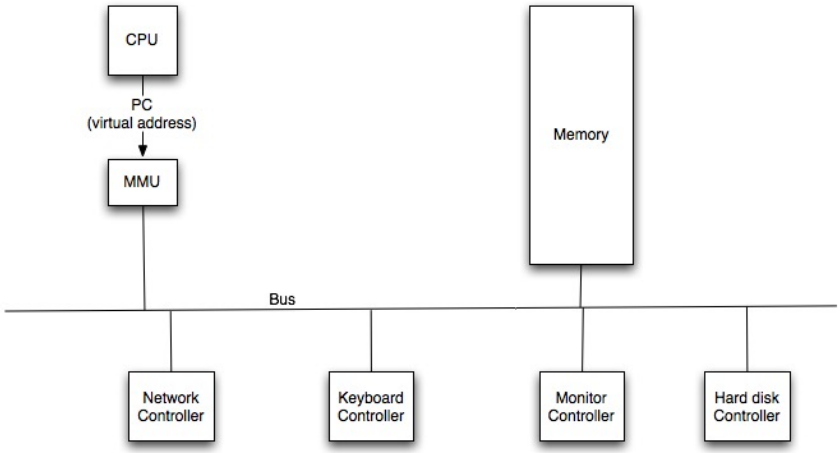


Figure 1.14: The general computer architecture for this book

1.4.3 Algorithm for loading a program

Now let's imagine that we would like to run the program in Figure 1.13 on our computer. As mentioned, in order to run a program, the program must be loaded in the main memory. Why? As we discussed before, the CPU operates by fetching the instruction pointed by the program counter register, decoding it, and executing it. That means the PC must point to the memory address of the beginning of the program. Let's say that our computer has 1K memory. Of course this is a really small amount of memory. At times, we envision such a machine with a small memory size to make it easier to

illustrate examples. Would such a machine be hard to imagine? In fact, Altair 8800 had 256 bytes of RAM in its first version. Furthermore, NASA's original Voyager has two processors, each having only 4K memory.

1K memory means 1024 bytes of words. We assume one instruction is 32 bits long, which is 4 bytes per instruction. Assume that the operating system is loaded from address 0 to 128-1, in hexadecimal 0x80-1. This means that we can load a user application program from the address 0x80. The operating system does the loading.

1.4.4 The concept of address space

When you compile, say, a C program, the compiler generates a binary executable code for that C program. To load the executable code to the main memory, OS creates an *address space* for the code. The address space more or less minimally consists of four different segments: the text segment, the static data segment, the dynamic data segment, and the stack segment, starting from the lower memory to the higher memory address. Since the compiler does not know where in the main memory a program will be loaded, it will generate the executable code as if it will be loaded from the memory address 0. All the relative addresses, such as target addresses for branch instructions, will be relative to address 0. We will discuss this concept further in the next section when we discuss how a program is actually loaded in the main memory. Now lets see how OS would load the program algorithmically. For now, it is sufficient to understand that each program that runs on a computer has its own address space. In general, the address space of one program is protected from other programs, which means no other program can access the address space of the program. Of course, the OS should be able to access the address space of user programs.

1.4.5 Loading an executable code to the main memory

Let's discuss what happens when a user double-clicks a program icon. The OS must first load the program to the main memory by creating the address space for the program. OS then will create all other necessary data structures for the program and schedule the program to run. We call the loaded program a *process.*

The OS can run Algorithm 1.16 to load a program. There are several other techniques to load a program in the main memory; we will discuss this topic in detail in Chapter 3. Figure 1.15 shows what the main memory should look like when the OS executes the algorithm successfully.

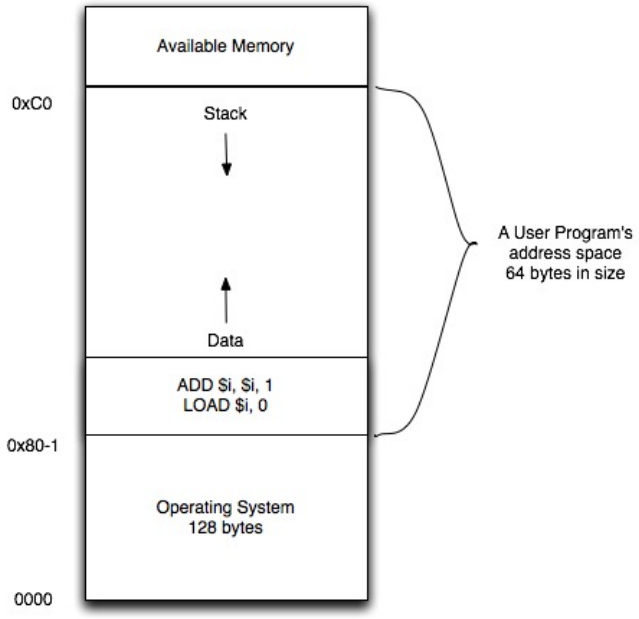


Figure 1.15: The resulting memory map when the algorithm in Figure 1.16 had been successfully executed.

```
boolean LoadProgramContinously () {
int usrProgStartAddress // a global variable
int usrMemorySize // a global variable
Procedure MemoryAllocate(exeFilePtr, codeSize) {
 if (codeSize> 0 && codeSize <= userMemorySize)
  ReadToMemory (userProgStartAddress, exeFilePtr,
                                              codeSize);
 else
  PrintErrorMesssage();
}}
```

Figure 1.16: A C program (part of OS) to load a program to memory continuously

Recall that the compiler assumed the program starts from address 0; that is the first instruction of the program to be loaded at address 0 of the main memory. However, in the example, the addresses range from 0 to 127

(i.e., 0x80 - 1) is occupied by the operating system itself. We do not want to load the user program from address 0. That will overwrite some part of the operating system, which will likely cause computer to crash.

Therefore, the OS keeps track of the next available memory address to load a new user program. In our example, it is 0x80, and the above algorithm keeps track of the address with the variable `usrProgStartAddress`.

Virtual address and physical address

Now we are ready to discuss the concepts of virtual address and physical address. These concepts are extremely important, and everyone must understand them before moving on to understanding memory management schemes.

One way to think about virtual address is the compiler view of program address. As we discussed above, a compiler always considers that the compiled program will be loaded from address 0 of main memory and the program is loaded continuously from address 0 to the address that can accommodate whatever size of the program. For example, if there is a program with size of 64 bytes total,[7] the compiler will assume the program will be loaded from address 0 of the physical memory and loaded until the address 63, without any discontinuity in the physical memory address. Loading a program to main memory without any discontinuity is called *contiguous memory allocation*. In Chapter 3, we will discuss various memory allocation schemes including non-contiguous memory allocation schemes. For now, we will assume our memory allocation schemes are all contiguous memory allocations.

Notice however, our example program in Figure 1.13 must be loaded from the address 0x80; it cannot be loaded from physical address 0 because doing so will overwrite the OS. This means that the virtual address 0 of the program is now mapped onto the physical address of 0x80. Assume the program is 8 bytes (i.e., 5 bytes for 5 instructions, 2 bytes for variables `sum` and `i`, and 1 byte for some extra stuff). Then we will load the program from 0x80 to 0x87. The virtual address 7 is then mapped on to 0x87.

> Checkpoint: If you are not familiar with the concept of virtual and physical addresses, you need to pause here and read the previous sections again.

[7] This is an extremely small program, but for the sake of the simplicity of explanation we use this small size

Exercise: Do Exercise 1.17 at the end of this chapter.

1.5 Concept of process and address space

In Section 1.3, we learned about what happens when we turn on a computer (cf. the boot-process). Now the computer is up and running, and the operating system has started. After logging in, if you envision a GUI-based system, a user can start a program by double-clicking the icon associated with the program. For example, if the user wants to start the web browser Firefox, she will double-click the Firefox icon. We all know that after the double-click, a Firefox browser window pops up on the screen. The question we should ask now is what happens in between? In other words, what are the things the OS must do from the time the user double-clicks the Firefox icon until a Firefox browser window pops up?

We briefly explained the concept of the address space; and an address space is created and loaded in the main memory when a program is loaded. But there is more to this.

First, for each program to run, a corresponding *process* is created by OS. A process is a *program in execution.* Let's define what a program is then. A program is, in our perspective, a source code written in C or any other high-level programming language or an assembly language. This means that a program can be stored in many different media including in a hard disk, in a magnetic tape, or in a USB memory stick. It can also be stored (or written) on a piece of paper although reading a program written on a piece of paper and loading it automatically to a computer would be much harder, but it is not impossible.

First, for each program to run, a corresponding *process* is created by OS. A process is a *program in execution.* Lets define what a program is then. A program is, in our perspective, a source code written in C or any other high-level programming language or in an assembly language. This means that a program can be stored in many different media including in a hard disk, in a magnetic tape, or in a USB memory stick. It can also be stored (or written) on a piece of paper. Although reading a program written on a piece of paper and loading it automatically to a computer would be much harder, it is not impossible.

On the other hand, a process is a program loaded in the memory that is to be executed by the CPU.

The following is roughly the series of steps that occur when a user double-clicks an applications icon (e.g., the Firefox icon) on the computer desktop.

1. User double-clicks the Firefox icon.

2. OS creates a pProcess control block (PCB) data structure for the program. The PCB contains many information about the process being created. Figure 1.18 shows a minimal data structure elements for a PCB.

3. OS creates the address space for the program and loads the program's address space according to the memory management scheme being used. Remember, we are only using contiguous memory allocation management scheme. Therefore, the virtual address space and the physical address space are related to each other in the same way as the ones we discussed previously.

4. OS then inserts the PCB to the *ready queue* of the OS. The ready queue contains all processes in the computer system at the time that are ready to run by the CPU. If we can load and run only one user program at a given time—that is not a multi-processing system—the ready queue will be at most contain one process.

5. OS then performs a *context-switching* to let the Firefox program run. While Firefox is running, OS is not running, assuming there is only one CPU.

I will try to clarify the above steps further now. Figure 1.17 shows what happens right before OS performs the context-switch in the above steps. Note that the ready-queue and the PCB are stored in the OS area of the main memory because they are data structures belonging to OS. The address space is stored in the user memory area in the prescribed way by the memory management scheme—in this example, a contiguous memory allocation scheme. Note also that in the above steps, until Firefox starts to run (i.e., a browser window pops up), the OS runs on the CPU. We assume there is only one CPU. Therefore, in order for Firefox to run, OS has to prepare Firefox to run on the CPU and then "step aside" so that Firefox can run. By "stepping aside" we mean relinquishing the CPU or equivalently performing a context-switching.

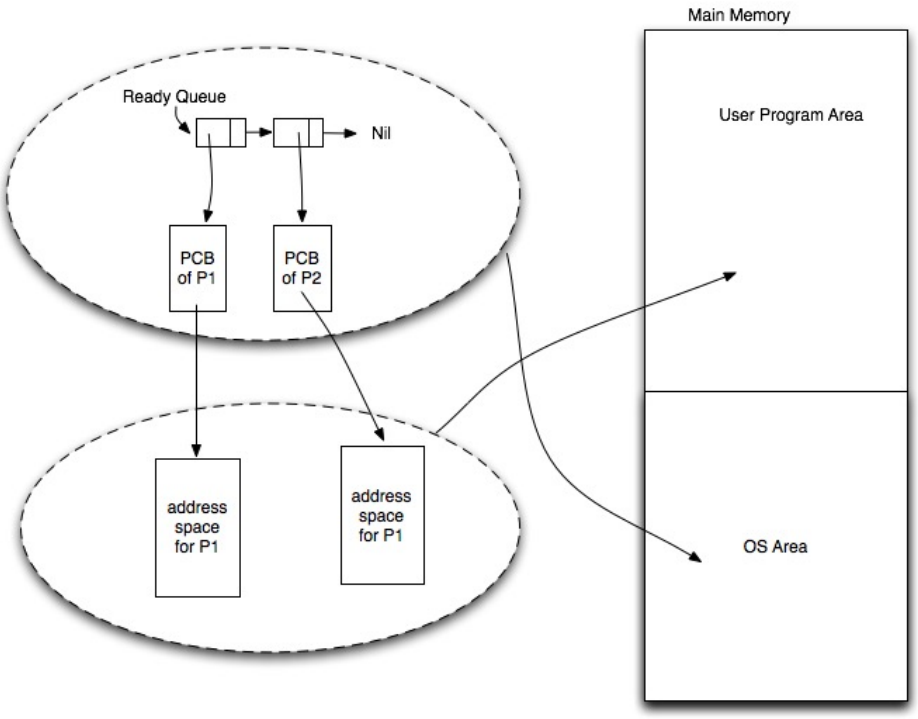


Figure 1.17: A data structure view of after a Firefox process, P2, has been created. There is another user process P1 in the ready queue as well.

```
typedef PCB {
 char* process_name;
 int process_number;
 int process_status;   // Ready, Run, and Blocked
 int program_counter;
 int stack_pointer;
 int registers [32];
 int* address_space; // ptr to the address space
                                  //(ptrs for segments).
 File* open_file_list;
 // other global information and management stuff
}
```

Figure 1.18: A minimal PCB structure

What is context-switching?

What then is context-switching? By *context* here, we mean by the CPU context, which is the CPU register state. The CPU register state, in simple terms, is the values of registers at a given time, including the values of the program counter and all general purpose and special purpose registers. Earlier we discussed what CPU does when it gets electrical power. The CPU clock will drive the CPU to the cycles of *fetch-decode-execute*.

The CPU will fetch the instruction stored in the address pointed by the program counter, decode it, and execute the instruction accordingly. How each instruction gets executed is a CPU design and microprogramming issue, and it is beyond the scope of this textbook. It is sufficient to understand that instruction executions are accomplished by the hardware circuitry in the CPU. Because the CPU will just "blindly" fetch the instruction stored in the address pointed by the program counter, if we want to run Firefox instead of OS, all we need to do is properly initialize the value of the program counter, which is the address of the next instruction to be executed. In this case, the address of the first instruction of Firefox must be stored in the program counter. Of course we need to initialize other CPU registers to proper values, such as initializing the general purpose registers to zeros, if Firefox runs from the beginning of its code. If Firefox was suspended for a while after running, and if we want to resume its execution, the PC and other registers would have to be restored with the values before the suspension. The PCB of Firefox keeps these values while Firefox is being suspended. For the purpose of understanding the context-switching from OS to Firefox, this is a sufficient simple explanation. In the current OS we are considering, because only one user program can run at any given time, there will always be context-switching happening between the OS and the user program (Firefox, in the example). See Figure 1.19, which illustrates a context-switching from the OS to a user program (Firefox, in this case).

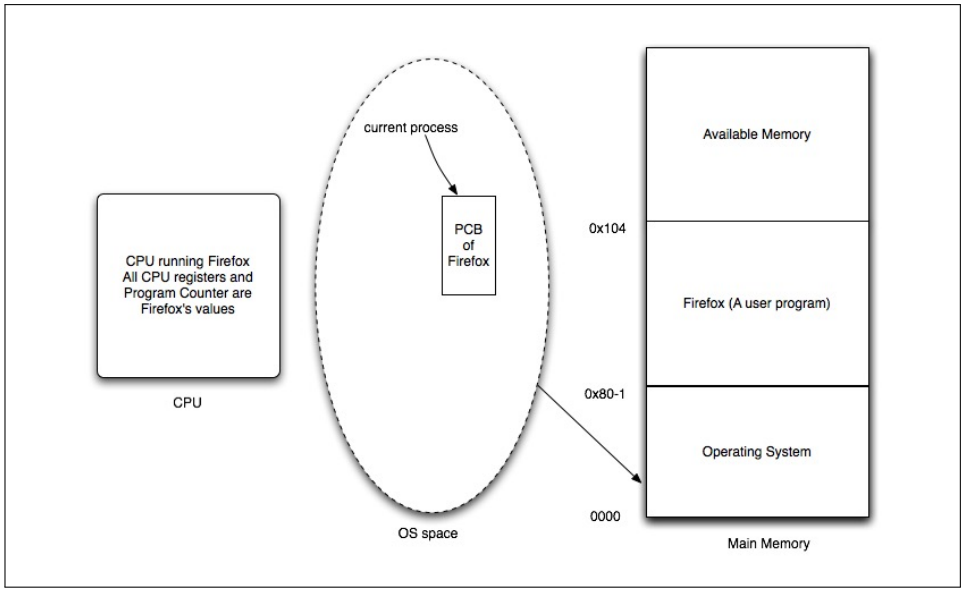


Figure 1.19: A simple context-switching from OS to a user program

When we start discussing multi-programming later, we will discuss more about context-switching.

1.6 Multi-programming

Multi-programming allows us to run multiple programs concurrently on a computer. We assume there is only one CPU; therefore, at a specific time, t, there is only one program running on the computer. Strictly speaking, multi-programming is an ability to switch from running one process to another without necessarily finishing the one that was running. The term is also analogous to *multi-tasking*. Therefore, the ability to perform the multiple work simultaneously by switching from one task to another.

In the previous sections, we assumed there were only two software entities running on our computer: the OS and single user program. There are no other programs running. Nevertheless, we need to be careful about the meaning of *running a program.* As we said, we cannot run multiple programs at any given specfic time, because we only have one CPU. When we say "running multiple programs," we mean that we load multiple programs in the main memory at the same time and perform context-switchings among them so that the human users would have an illusion of running multiple programs at the same time; this is because context-switching is fast so that

humans feel that all these programs are running at the same time in parallel. However, they are not.

Why then multi-programming is needed? The following example should motivate us. Consider the following three programs in Figures 1.20, 1.21, and 1.22. The `Hello World` program prints out the message "Hello World" a million times, the `I am infinite` program just repeats the while-loop infinitely, and the I/O program awaits a user's keyboard input and prints out the number of characters in the input the user typed in.

1.6.1 I/O bound tasks

```
#include < stdio.h>
Hello_world() {
   int i;
   i = i + 1;
   for (i =0; i < 1000000; i++)
        printf ( Hello  World );
   exit(0);
}
```

Figure 1.20: A Hello World Program. This is an I/O bound job.

`Hello World` doesn't perform much computation or utilizes the CPU. However, it spends most of the time printing out the messages to the screen. In other words, the program spends most of its time doing output. Again, this means that it doesn't use the CPU most of the time. An example of this type of program would be a program that prints out payroll checks for all the employees for a company. We call this type of programs I/O bound tasks. In this particular case, it is an output-bound task.

1.6.2 CPU bound jobs

```
I_am_Infinite() {
 while (1);
}
```

Figure 1.21: A program with an infinite loop. This is a CPU bound job.

The `I am Infinite` program on the other hand has no output or input. It just runs forever. This program within its while-loop will keep using CPU cycles. No other computing components, such as the monitor, hard disk, or keyboards, are needed to run this program. Although we gave an example in which the while-loop is empty, we can imagine that within the while-loop, the program may perform heavy computations, such as solving complicated mathematical formula. An example of this type of program can be a scientific program that computes weather predictions running complicated prediction algorithms. We call these types of programs CPU-bound tasks.

1.6.3 Interactive tasks

```
#include < stdio.h>
void read_from_keyboard() {
   int c, nc = 0;
   while ( (c = getchar()) != EOF )  nc++;
   printf("Number of characters in input
                            line = %d\n", nc);
}
```

Figure 1.22: An interactive job

The program in Figure 1.22 will wait for a user input, listening to the keyboard, then compute the number of characters in the input line using the CPU and print out the result using the monitor. We call this type of program an *interactive task* because it interacts with human users. Web browsers can be considered as interactive tasks.. For interactive tasks, the response time

is one of the most important performance metric. The OS must make sure interactions with users are quick.

Although these three programs look simple, in fact, these programs capture the characteristics of most of the existing computer programs (if not all). One of the main reasons for multi-programming is to utilize all parts of the computer systems as much as possible. If we run only one type of program, only certain parts of the computer system will be well-utilized. For example, if we only run CPU-bound jobs, CPU will be busy, but all other hardware components, such as keyboard, hard disks, monitor, etc. will be idle. When these three types of programs are well mixed, OS will be able to utilize all hardware resources. We will come back to these programs in the next section.

1.6.4 Process states

Now we have learned that a process is a program in execution. We also know that a process cannot run continuously on the CPU, due to activities such as I/Os. When a process is actually running on the CPU, we say the process is in the "running" state. If it is not running on the CPU and is waiting for, for example, an I/O to be completed for it, we say the process is in the "blocked" state or in the "wait" state. Sometimes we say it is "suspended." When a process is not running on the CPU and not waiting for an event as an I/O to occur but waiting in the ready queue until the CPU is available for it, we say the process is in the "ready" state. It means it is ready to run on the CPU. In other words, it means that the process, at that point of its execution, needs the CPU to run the next line of the code in the program. When a process is just created, we say its state is "just created." This state is useful when implementing the process in OS, but conceptually we can think of the "just created" state the same as "the ready state." A process that has been just created must be ready to run. That is, the first computing resource it needs is the CPU.

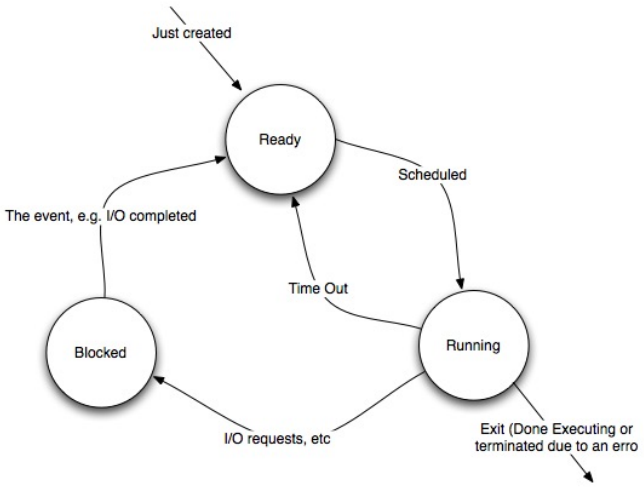


Figure 1.23: Process states

Figure 1.23 shows the transition of process states. When a process is first created, it is inserted in the ready queue. The scheduler (it's a subroutine in OS) will pick one of the processes in the ready queue when the CPU needs a program to run. The currently running process has two reasons to become blocked. First is that it calls a system call, such as an I/O request by, for example, executing read() or write(). The second reason is that in a scheduling algorithm called round-robin, each process is given a certain amount of CPU time (about some milliseconds) and then inserted back in the ready queue to give a chance to the next process in the ready queue to run on the CPU. This way, no one process can run forever on the CPU. To make this type of scheduling possible, we need a timer. A timer is a hardware device that generates an interrupt to the CPU for every designated amount of time. We will discuss this in detail in Section 1.6.5. A blocked process awaits in a queue. Let's call it a blocked queue. When the event it has been waiting for is completed, an interrupt notifys the CPU that the event has been completed. The the blocked process now can be inserted back to the ready queue by the interrupt service routine.

1.6.5 Timer and timer interrupts

A timer is a hardware device that ticks. A timer contains a quartz or something similar to keep regular ticks. A timer can generate a signal for every designated amount of time. Figure 1.24 shows an abstraction of how

the timer device is connected to the CPU. There is an interrupt pin in the CPU. When this pin's voltage goes high, the CPU transition to the interrupt state. Recall that the CPU is nothing but a finite-state machine. This CPU interrupt pin is connected to many other devices that can generate interrupts. These devices are typically I/O devices, such as hard disk controllers, keyboard controllers, etc.

After the CPU's state becames the interrupted state, it will find which device caused the interrupt—in our example it is the timer—and execute the interrupt service routine (ISR) in the OS. In our example, it will run the time interrupt service routine. To implement a round-robin scheduler, the timer ISR will call the scheduler subroutine. The scheduler subroutine will pick one of the processes waiting in the ready queue to make it to run. The timer interrupted process is inserted back in the ready queue for its next turn for CPU time.

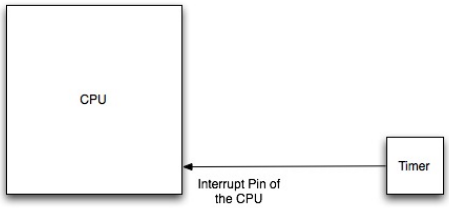


Figure 1.24: Timer and CPU. The timer interrupt signal pin is connected to the CPU's interrupt pin.

1.6.6 Why do we need operating systems?

An operating system has numerous jobs. But in essence, it supports user programs' needs so that user programs can run smoothly and efficiently. A computer system consists of numerous hardware components: the CPU, main memory, monitor, keyboard, hard disk, printer, etc. One of the jobs of an OS is making sure all parts of the computer systems are utilized well. For this reason, some people call OS a resource manager. Another job of an OS is to hide hardware or low-level details from user programs.

As discussed before, there are various types of users of computer systems. Roughly speaking, the user programs are programs used by human users—directly or indirectly—and these are developed by programmers. Programs such as Firefox and word processors are user programs. Although developing programs like Firefox requires a good amount of computer knowledge and programming skills, these application programmers do not know or do not

want to know how the hard disk is working. Examples include how the motors inside of hard disk work and how to keep track of which part of hard disk is free or being used. OS keeps track of these and makes a high-level interface to user programmers so that they don't have to worry about detailed hardware operations of hard disk. In this regard, people call OS as an abstraction-level manager, which means it abstracts away the messy details of low-level stuff from the application programmers. For example, a programmer may want to write a code that can read and write from and to hard disk. Most of the people who are reading this book already may know at least how to write some basic programs.

Consider the following C program in Figure 1.25. In the program, the commands `fopen(), fprintf(), rewind(),` and `fscanf` are all system calls. In particular, they are systems calls related to files. So we call them "file system calls." There are other types of system calls, such as process system calls, communication system calls, device management system calls, etc. System calls hides details.

```
#include <stdio.h>
int main() {
FILE *fileptr;
char buffer[100];
int i;

fileptr=fopen("os-data.txt", "w+");
for (i = 0; i < 7; i++) {
    fprintf(fileptr, "Hello\n");
}
rewind(fileptr);
fscanf(fileptr, "%s", buffer);
printf("What I read is %s\n", buffer);
}
```

Figure 1.25: A program with I/O system calls

Now we know that OS is trying to utilize all component of the computer system as busy as possible (utilized well), let's go back to our three types of programs presented above.

If we assume we only can run one program at a time (which means start the program and finish the program before running the next one), while running the CPU-bound job, all other parts of the computer system, except

the CPU, will be idle. On the other hand, if we only run the `Hello World` program, most of the time the monitor will printout the message and a little of CPU cycle will be used to check the number of times the for-loop has been iterated. The I/O bound program will let CPU idle even more because it will be waiting for a human input through the keyboard. For a CPU, this is like a millennium.

Then, how do we make all parts of a computer system as busy as possible? By multi-programming! Multi-programming means that we load multiple programs in the main memory and switch the processes in and out of the CPUs as they are needed or not . This means that we are allowing a program being switched out of the CPU in the middle of its execution and switching it back on to run on the CPU when it can. While an I/O is being done by a I/O device, we can run another program on the CPU.

To illustrate this, let's consider a multitasking process being done by a typical college student. John is a college student. He is taking three computer science courses: data structures, discrete math, and computer organization. This week, he has homework assignments for all three courses due next week. If we assume, mono-programming analogy, John will start with one of the homework assignment, say discrete math homework, and he will finish it before starting any other two homework assignment. Then choose the computer organization, and then data structures homework assignments. This type of scheduling is sometimes called batch processing. Technically Speaking, a pure batch processing system runs a process and finishes it before taking the next one. However, this method may not be efficient. For example, after starting the discrete math homework one third of the way, John realized that he need to go to the library and find a paper on some subject before making more progress on the discrete math homework. However, the library is closed until three days later. What would John do? He would do what most of good students would do. He would remember where he left in the discrete math homework then start another homework assignment until he can go to the library. So he starts the computer organization homework. This will let John productively utilize the days before he can go to the library. The same thing can be true for the computer systems.

Considering the three programs again

Now we go back to our three original example programs. Let's denote for convenience that p1, p2, and p3, refer to Hello World, CPU-bound, and interactive process, respectively. In a batch system, when p1 needs to print out the message "Hello World," the CPU is idle. This is because for modern

OS, CPU is not involved in I/O directly. There is a rudimentary I/O method in which CPU has to be involved in every step of I/O process. We call this method *programmed I/O*. Read Section 1.6.7 for three different I/O methods. When p1 is done, the computer can run p2, then p3. For example, when p2 is running, all other components in the computer system except for the CPU are idle. When p3 is running, when it is waiting for the user input from keyboard, all other components of the computer system are idle except for the hardware related to the keyboard.[8]

Now as we can see, if we only run one program at a time and finish the current one before starting the next program, only one component of the computer system is utilized at any given time. This because a program will need to use CPU sometime, then the monitor, and then a printer perhaps, but never more than one component at a given time. Anyone who programmed a computer knows that a program is executed line by line; and each line of a program may be a computation, or an output, or an input, etc.

Therefore, if we want to make multiple components of computer system busy at the same time, we load multiple programs in the main memory and run them concurrently. If a program is needing the CPU, run it on the CPU; if it is done using the CPU and now trying to execute the program command that prints out something on the monitor screen (i.e., an I/O request), let it relinquish the CPU for that period of printing so that the next process that has been waiting to use the CPU can use the CPU. Where would these processes waiting to use CPU wait? It's the ready-queue we mentioned before.

The ready-queue is a queue of processes that are ready to run on the CPU. What does it mean to be ready to run on the CPU? This means that the next instruction (or program code) to execute is a computational instruction. For example, in Figure 1.26, when the program's code `sum = sum + 1` is to be executed, the program needs the CPU because the code is computational and needs the CPU. However, after the CPU is done executing the code, the next one to be executed is `printf("Sum is %d\n", sum);`. This is an output command and therefore cannot be executed by the CPU. An I/O (in this case, output) must be scheduled so that the message can be printed out to the monitor screen. Meanwhile, the CPU can be idle. Instead of the letting the CPU idle, the OS, specifically the *scheduler*, fetches the next process waiting in the ready-queue and lets it run on the CPU.

[8] In fact even the keyboard hardware would be idle because the hardware is waiting for a key being pressed to generate an interrupt. We will discuss about the interrupts later.

Meanwhile, the process that requested the output has to wait in an I/O wait-queue until the output is done. When the output is done, the output device will generate an interrupt, the OS then will run a proper interrupt service routine to put the waiting process in the ready-queue, because its I/O request has been completed. Figures 1.30 to 1.34 show the dynamics.

```
int main() {
  int i = 0;
  int sum = 0;
  for (i = 0; i < 5; i++)
      sum = sum + 1;
      printf("Sum is %d\n", sum);
}
```

Figure 1.26: C program to illustrate the ready-to-run concept

1.6.7 The compilation process and three I/O methods

Computer programs should have the ability to communicate with the outside world. They need to be able to accept the user's inputs and be able to output its responses. It is not hard to imagine that virtually every program we encounter has this capability. Can you imagine a web browser without an I/O capability? I cannot. For this very reason, programming languages such as Java, C, and Lisp, all include I/O capabilities. Strictly speaking, the I/O libraries are not part of the core language syntax. This is due to the fact that programming languages are hardware platform independent, while I/O routines are hardware dependent.

The compilation process

The fact that I/O library routines are not part of many programming language's core affects how the compilation process is designed as in Figure 1.27. In the figure, `helloworld.c` source program is an input to gcc (GNU C compiler). First it generates an intermediate file `helloworld.i`. This file is still a text file yet resolves macro definitions and others in the original source file. For example, if the original source contains a statement like `#define LIMIT 10`, each instance of `LIMIT` in the source will be replaced by '10' in `helloworld.i`. Step 3 will compile the program and generate an assembly program that is equivalent to the original program. However, I/O

calls and other library calls are not resolved yet. In other words, the binary executable part of these calls are not yet part of the assembly code. The code optimizer will optimize the assembly code. Finally, the optimized code is now linked with binaries for the I/O and other library calls to produce `helloworld.o`. If there are other `*.c` source programs to be linked with `helloworld.o`, they will be linked together in this step. This is because that there can be several `*.c` source programs that go together. Finally `a.out` or `helloworld`, depending on the output file name specified, is produced. That is the final executable binary code. In Windows, it would be `helloworld.exe`.

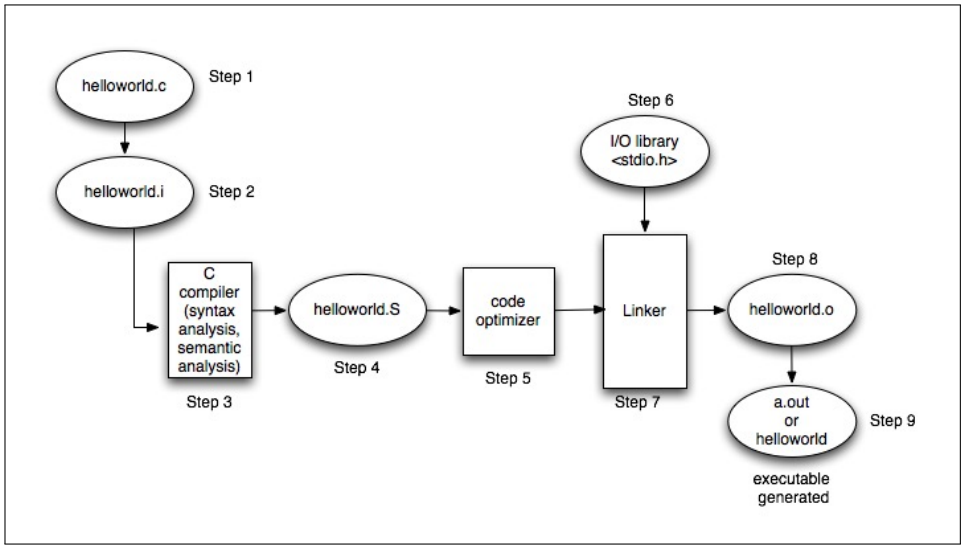


Figure 1.27: The compilation process for gcc compiler. This process is more-or-less similar in almost any complied programs. Note especially that the linker links pre-compiled I/O libraries and other libraries to the binary code are generated in the previous step. This step resolves all the I/O and other library calls and makes a complete executable binary.

The I/O methods

There are generally three I/O methods: (1) Programmed I/O; (2) Interrupt-Driven I/O; and (3) DMA I/O, in the order of sophistication. What I mean by "sophistication" is how much the method frees up the CPU to do other things while an I/O is happening.

In earlier computing systems and perhaps even in recent systems that

are relatively simple, programmed I/O methods are used. Recall that a CPU is a device that can do computation such as multiplication, subtraction, logical operations, etc. However, any I/O operations must be done by the corresponding I/O device. Every I/O device has an I/O controller (an electronic device) associated to it. A hard disk device has a hard disk controller, a monitor has its controller device (usually it is a graphics card), and a printer has a printer controller. These controllers are circuitry that understand how to interact with the corresponding I/O device. Usually, each I/O device comes with a driver program. The driver program is usually written by the I/O hardware device's manufacturer because the company knows the hardware of the I/O device the best. Therefore, from the I/O device (hardware), we have a controller (hardware), the device driver (software), and the OS (software). Of course the OS interacts with application programs.

A controller has at least the following hardware components: the command register, the data register, and the status register. The command register can be used to store the command for the I/O device connected to the controller. For example, if we want to read from a hard disk, we write the command `'read m'`, where m can be the disk address, to the command register and write the memory address of the buffer for the data to be transferred from the disk to the memory. This will cause the controller circuit to drive the hard disk. Note that the buffer is owned by the program that requested the read operation. Of course the commercial grade controllers are much more complicated to improve the performance of the I/O devices connected to them.

To illustrate, consider the C program in Figure 1.28 that writes messages to the monitor.

```
#include <stdio.h>
int main() {
int i;
for (i = 0; i < 7; i++) {
    printf("Learning OS is fun\n");
}}
```

Figure 1.28: C program that writes to the monitor

To execute `printf(fout, "Learning OS is fun\n");`, a system call to the OS is made. Then OS will execute the appropriate steps to fulfill the I/O request for the user program.

In programmed I/O, the CPU is involved with the I/O process in every step. Once the command `printf(fout, "Learning OS is fun\n");` is made, OS will start to run, then OS will execute a subroutine that does something similar to what is shown in Figure 1.29. The subroutine sends one character at a time to the output device's data register and interacts in a handshaking way. Notice that the CPU must run this subroutine while the I/O is being done. Consequently, the CPU cannot do anything else. Because I/O devices are much slower than the CPU, the entire process will take quite a bit of time from the CPU's perspective.

```
programmed_io (char[] data_output) {
 for (i = 0; i < length(data_output); i++) {
       copy_char_to_data_register(data_output[i]);
       set_up_other_registers();
       wait_for_io_done();
  }}
```

Figure 1.29: A pseudo-code for programmed I/O as a subroutine of OS

In order to free the CPU from I/O duties, the interrupt-driven I/O has been developed. In this method, after an I/O request is made, the OS blocks the I/O requesting user program P1, as in Figure 1.30. As the result, we see Figure 1.31. Then OS can schedule the next process, P2 in the ready-queue, on the CPU while the I/O is being performed by the I/O device. When the I/O is done, the I/O device will generate an interrupt to the CPU to notify that the I/O for P1 is done so that P1 may continue to run. Caution here though. P1 will not immediately run after its I/O request has been served. Instead, it is inserted back in the ready-queue so that the scheduler (part of OS) may choose P1 according to the scheduling policy (Figure 1.34). We will discuss scheduling policies in Section 2.5 of Chapter 2.

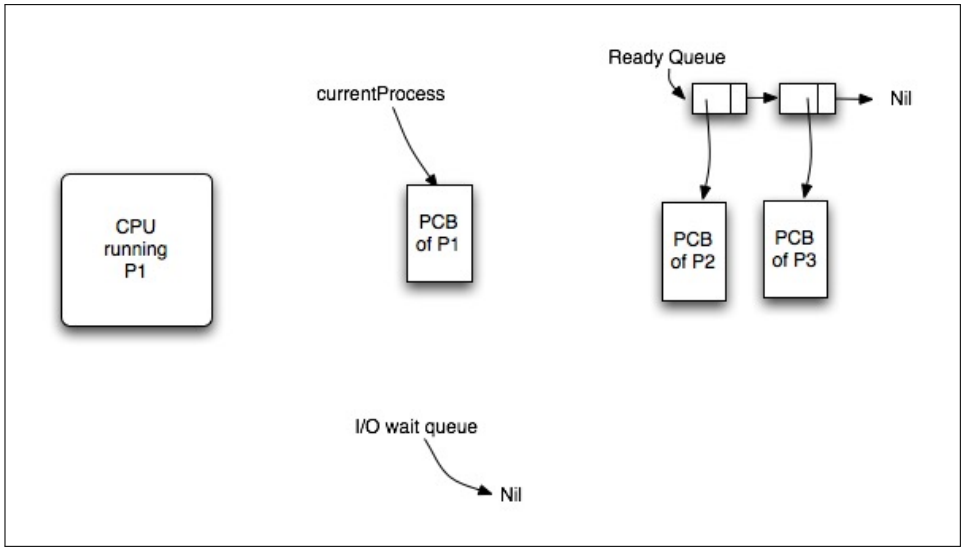


Figure 1.30: Data structures just before P1 requests an I/O. i.e., before executing the printf() function

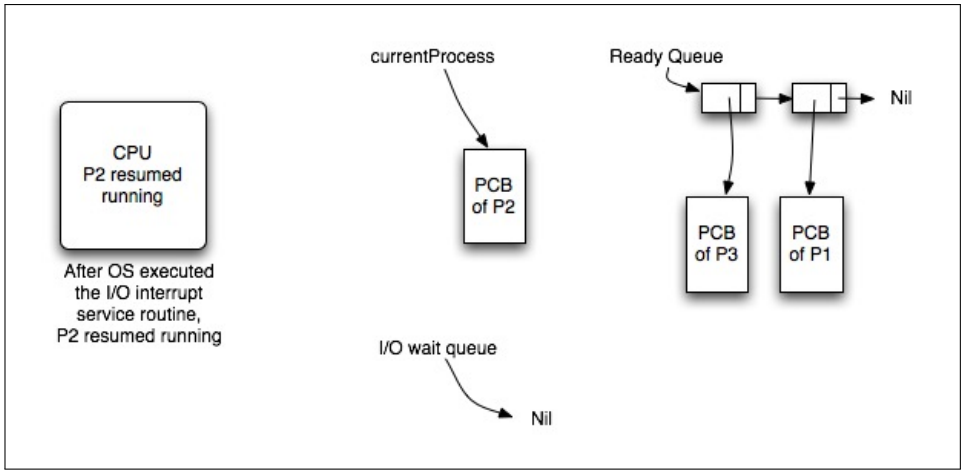


Figure 1.34: Now P2 resumes to where it left off just before the interrupt, it continues to execute. Notice that now the I/O wait queue is empty.

The interrupt-driven I/O method definitely frees up the CPU from getting involved in the I/O process. However, for example, if a read() request

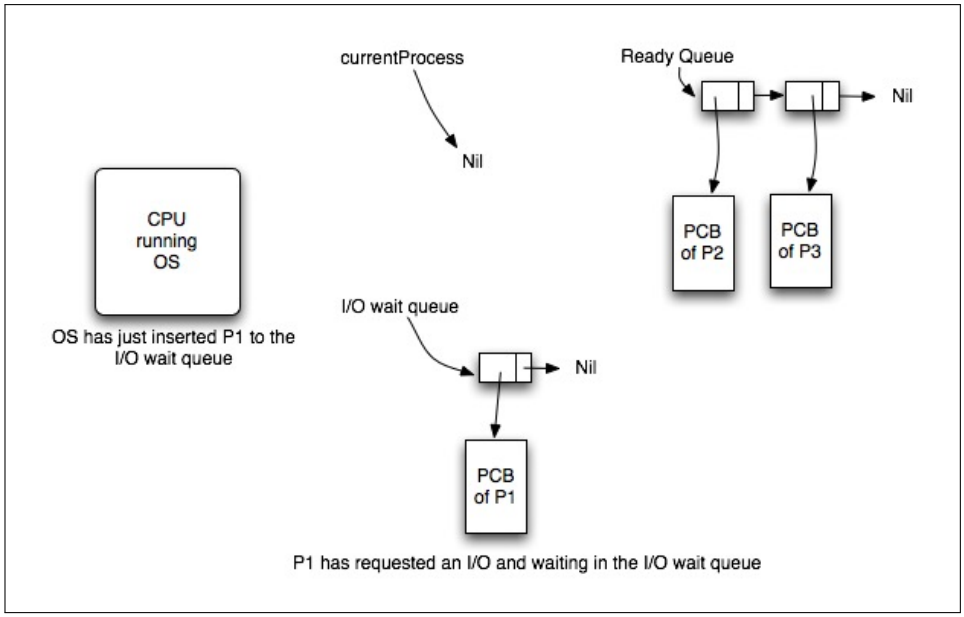


Figure 1.31: Data structures right after P1 requested an I/O. i.e., right after making a request to execute the printf() function

is made, the data read from, say, hard disk, will be first brought into the OS memory area. Therefore, it must be copied to the user program's address space (i.e., P1's buffer address). That copy still needs to be done by the OS.

The DMA I/O method will free the CPU from such a copying operation. In order to have the DMA I/O method, the computer system must have a DMA controller (a hardware component) that does the all the details of I/Os for the CPU. One important thing to remember is that the DMA I/O method still uses interrupts. It is essentially the same as the interrupt-driven I/O except that now the interrupt is served by the DMA mostly. Just think about the DMA controller as a designated processor that specializes I/O operations, whereas the CPU is a designated processor that specializes computations.

1.7 The user mode and the privileged mode

By now, you should realize that computers run roughly in two different modes: the user mode and the privileged mode. When the CPU of a com-

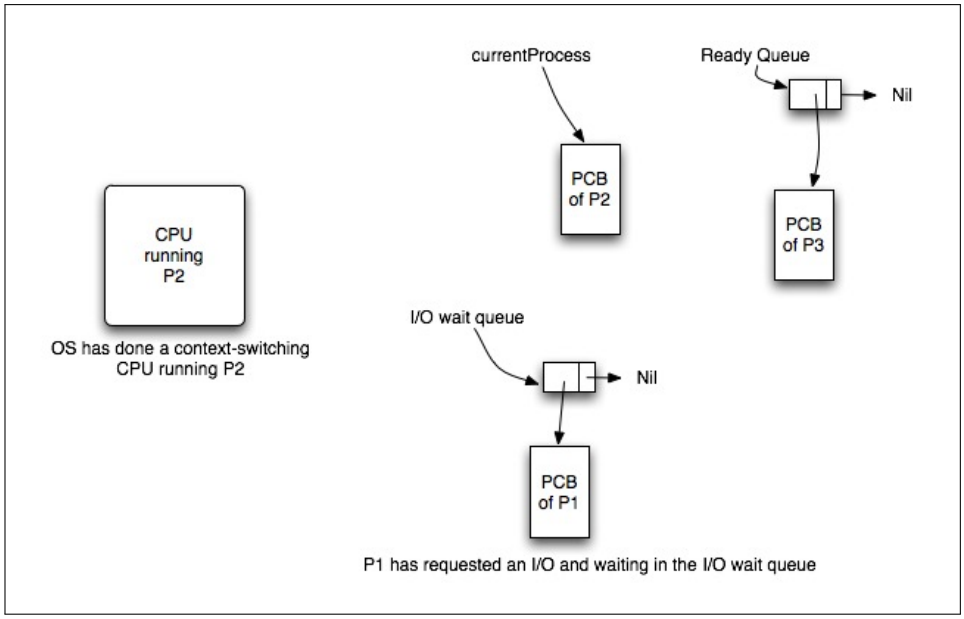


Figure 1.32: Data structures while P1 is waiting for printf() function gets fulfilled. Since P1 cannot proceed until the `printf()` is done, it is waiting in the I/O wait-queue. The OS, specifically the scheduler schedules P2 on the CPU so P2 is running on the CPU now. Notice the `currentProcess` pointer and the content of the ready-queue now.

puter runs a user application program, we say the computer is in the "user mode." User application programs are programs we use every day, such as web browsers, word processors, email clients, etc. When the CPU runs the operating system, we say that the computer is in the "privileged mode" or in the "protected mode." Some people use the term "kernel mode" for this mode as well. In the privileged mode, the computer should be able to access all or most of the hardware devices without much restriction. When the computer is running in the user mode, the CPU and OS restricts what it can do so as to protect hardware and software systems properly.

1.7.1 Traps, system calls, and interrupts

There are three different ways for the CPU to transition from the user mode to the privileged mode. They are *traps, system calls,* and *interrupts.*

Traps are generated when some types of errors occur during an execution

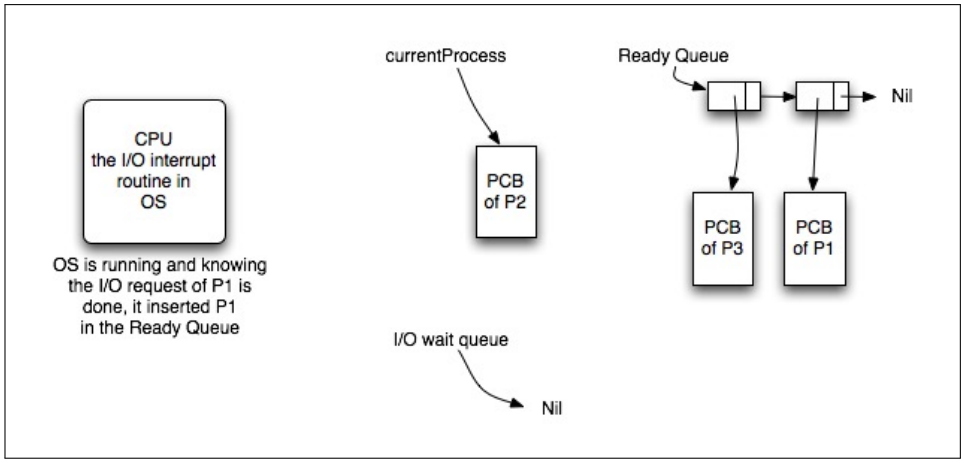


Figure 1.33: Data structures right after `printf()` function has been fulfilled by the I/O device, in this case the monitor. The monitor's control card (graphics card) just generated an interrupt to the CPU. After getting the interrupt signal, CPU suspends the execution of P2 and starts running the I/O interrupt service routine, which is a part of OS. The I/O interrupt service routine inserts P1 to the ready-queue (because it is done with the `printf()` statement, and now it is ready to execute the next line of the program.

of a program. For example, when the CPU runs a user application program, if it encounters a division-by-zero error, because the program cannot continue anymore, OS must take care of the situation. When a CPU trap is generated, a subroutine in OS takes care of the situation properly. Traps are also called *exceptions.* Another type of a trap is invalid memory access. When a user program attempts to access an invalid memory outside of its address space, a trap is generated. OS will most likely kill the user program process.

We have already discussed interrupts. Interrupts are generated by external devices such as hard disks, key boards, etc. When, for example, a keyboard stroke is detected by the keyboard controller logic, it generates an interrupt signal to the CPU. When the CPU gets an interrupt signal, it will stop what it has been doing and jump to a subroutine that can take care of that specific interrupt type. We call such a subroutine, interrupt handler, or interrupt service routine.

Systems calls are another way to make a transition from the user mode

to the privileged mode. System calls are made intentionally by the programmer. For example, when the currently running program calls a read() function, a proper system call subroutine in OS needs to be called to serve the request from the program.

1.8 Wrapping up

We have covered a lot of materials in this chapter. We looked into an entire view of computer operating systems. This chapter is a long one; it is important to understand how various components in OS and hardware are interacting. If we only understand each component in isolation, we don't understand how OS works. It is always important to relate a new concept you learn to the ones you already know. In the following sections, we will go into details of the concepts we have already discussed.

Exercises

Ex. 1.1 — There are several different levels of abstraction for each type of users see the computer system. Consider Bob, who uses computers for word processing, and Alice, who uses computers to develop a program such as web browsers and word processors. Also consider Andria, who use computers to study methane-eating bacteria. How do they interact with the operating systems and how can operating systems designers help each person for their work?

Ex. 1.2 — Get together with about five of your classmates to discuss the following. How would learning about and understanding operating systems make each of you a better application programmer? Summarize the discussion.

Ex. 1.3 — In executing i = x + 1 instruction, what would be the two input to the ALU depicted in Figure 1.3?

Ex. 1.4 — What is a cross-compiler?

Ex. 1.5 — What is the role of the magic number for binary executable files? Where is it stored?

Ex. 1.6 — List all segments that consist an address space and explain each segment.

Ex. 1.7 — Discuss other improvements we can do for Altair 8800.

Ex. 1.8 — What are von-Neumann machines, and how are they related to the computers we use today?

Ex. 1.9 — Anyone who programmed in Unix environment using C or C++ (or other languages as well) surely experienced the famous error message "segmentation fault (core dumped)." Explain what this error message mean and list possible reason for this error. What does "segmentation" mean in this context? What does "core" mean in this context?

Ex. 1.10 — Redesign this: Finding the physical address from a virtual address in a contiguous memory allocation scheme. Give a virtual address, how would you find the corresponding physical address if you know the first physical address of the program to be loaded? Give a mathematical formula.

Ex. 1.11 — When a compiler generates the binary code for a source program written in a high-level programming language, it does not know where

and how the binary code will be loaded by the operating system. Why not? In order to generate a binary code, the compiler, however, must make certain assumptions on where and how the binary code will be loaded in the main program by the operating system. What are the reasonable assumptions that are made by most of compilers in terms of where and how?

Ex. 1.12 — What happens when we double-click a program icon? Or type a command at the prompt? Describe all the steps happening in OS until we see the application on the screen.

Ex. 1.13 — A program can also be stored (or written) on a piece of paper although reading a program written on a piece of paper and loading it automatically to a computer would be much harder, it is not impossible. In order to read and load a program written on a piece of paper automatically to a computer, what kind of peripheral devices are needed?

Ex. 1.14 — Altair 8800 had limited hardware and software. Programming was difficult. In fact, the programmer used flip switches to load and run the program. Bill Gates and Paul Allen wrote a Basic programming interpreter for this machine. How would this interpreter can help make programming easier?

Ex. 1.15 — Discuss the steps followed during the boot process in a step-by-step manner.

Ex. 1.16 — Z1, developed by Konrad Zuse around 1936, is known to be the first programmable binary digital computer. Learn about this computer and discuss the similarities and differences of Z1 and the modern computers. Also, explain how programming was done in Z1.

Ex. 1.17 — Finding the physical address from a virtual address in a contiguous memory allocation scheme. Given a virtual address, how would you find the corresponding physical address if you know the first physical address of the program to be loaded? Give a mathematical formula.

Ex. 1.18 — The term context-switching is used to describe switching from one program to another in execute. In here what is the "context" being switched?

Ex. 1.19 — Why do we need to make sure all three types of processes, I/O bound process, CPU bound process, and interactive processes are well mixed in scheduling?

Ex. 1.20 — What is the *round-robin* scheduler?

Ex. 1.21 — Process control block represents the data structure for process. Each process created by OS has its own PCB. In Linux, PCB is implemented in the `struct task_struct`. Read about this on the Internet. Also find an educational OS such as Nachos or MINIX and find the corresponding PCB implementation. Discuss what you have found.

Ex. 1.22 — A timer is a hardware device. It can generate an interrupt signal at a predefined time interval. Explain how a timer device is used for the *round-robin* scheduling in multi-tasking operating systems.

Ex. 1.23 — A running process can be blocked when certain event occur. Explain all events that make the current process (i.e., currently running process on the CPU) go into the blocked state.

Ex. 1.24 — Explain differences between the user-mode and kernel mode. Explain this from the CPU hardware's perspective.

Ex. 1.25 — Study the boot process of Linux system. Describe what Master Boot Record is.

Ex. 1.26 — What are the meta-data items to be considered in file system design?

Ex. 1.27 — What are the meta-data items to be considered in process/thread system design?

Ex. 1.28 — What are the meta-data items to be considered in memory management?

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[3] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[4] Frederick P. Brooks, Jr. *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[5] Edsger W. Dijkstra. The structure of THE-multiprogramming system. *Commun. ACM*, 26(1):49–52, January 1983.

[6] Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, HOTOS'07, pages 20:1–20:6, Berkeley, CA, USA, 2007. USENIX Association.

[7] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[8] Per Brinch Hansen. The nucleus of a multiprogramming system. *Commun. ACM*, 13(4):238–241, April 1970.

[9] Per Brinch Hansen. *Operating System Principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1973.

[10] IBM. Ibm accounting machine types 402, 403, and 419: Principles of operation. Technical Report Form 22-5654-11, IBM, 1949, 1951, 1953.

[11] IBM. The ibm card-programmed electronic calculator model a1 using machine types 412-418, 605, and 941: Principles of operation. Technical Report Form 22-8696, IBM, 1954.

[12] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice Hall Professional Technical Reference, 1984.

[13] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[14] Marshall Kirk McKusick and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2004.

[15] MITS. MITS Altair Basic. Technical report, MITS, Inc, 1975.

[16] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, 1974.

[17] Saul Rosen. Electronic computers: A historical survey. *ACM Comput. Surv.*, 1(1):7–36, March 1969.

[18] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts 7th Edition with Java 7th Edition*. John Wiley & Sons, 2006.

[19] W. Richard Stevens and Stephen A. Rago. Advanced programming in the unix environment. *SIGSOFT Softw. Eng. Notes*, 38(6):45–45, 2013. Reviewer-Teodorovici, Vasile G.

[20] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[21] Uresh Vahalia. *UNIX Internals: The New Frontiers*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1996.

2
Processes and Threads

In Chapter 1 we said a process is a "program in execution." Loosely speaking, any program that is currently running has at least one or more processes. At the beginning of a program execution, only one process is created for the program in general. As the process continues, more processes can be created using special system calls. In Unix-like systems, `fork()` is used for creating a new process. When you double-click on the Firefox icon, it will start a process that runs the Firefox program.[1] If you are familiar with the object-oriented programming concept, a program (such as Firefox) can be considered as a class, and a process is an instance of the program. This is not an accurate analogy but may "ring the bell" for some people. Just as you can create as many instances of a class, you can make as many instances (i.e., processes) of a program, at least in theory. Unfortunately (in my opinion of course!), some of common operating systems, such as Windows, do not allow multiple instances of a program to be created. For example, if you double-click on Microsoft Word for the second time while it is already running, the OS won't start a new instances of Word. However, Windows

[1] In traditional Unix-like systems, in fact, if you click on the Firefox for the second time, it will create another Firefox process. The OS will allow the computer user to create as many Firefox processes as the user wants as far as there are enough of necessary computer resources, such as main memory, etc. More recently, OS developers started to tie web browsers to OS. Therefore, many OSs may create one process for a program and create multiple threads within the process as needed.

will allow multiple documents to be edited at the same time. Each document window can be associated by a thread. We will clarify the differences between processes and threads and benefits of using threads in Section 2.6.

2.1 Concepts of process

You may heard of concurrent programming or parallel programming before. When you write a program with a programming language that allows programmers to create multiple processes explicitly, users can create as many processes as you want to allow, as far as the OS allows. Being able to create multiple processes is empowering and convenient. This allows multi-tasking. Users may want to run multiple programs simultaneously to improve productivity, for example. Users may want to run a word processor and a web browser concurrently, for example. Each program run is associated with a process in general. Remember, a process is a program in execution. There are several ways that a new process is created.

Several ways for a process can be created

- A human user explicitly commands the OS to run a program by double-clicking the program icon or typing a command.
- A currently running process creates a sub-process (also known as a child process) prompted by an external event (a human user's request, an event caused by another process).
- A currently running process creates a child process explicitly executing an instruction such as `fork()`.

Readers may have already noticed that above three cases are fundamentally the same. In particular, the third case is a general summary of the previous two cases. You will explain why this is the case in the exercise at the end of the chapter.

2.1.1 Creating a process in Unix

In Unix, a common way to create a process in a program is using the `fork()` system call. Figure 2.2 shows an example of creating processes in the C programming language.

The fork() system call will create a new process (called a child process) that is an exact copy of the parent process. The function returns the id of

the child process that was just created to the process that created it (i.e., the parent process). The readers should wonder why the fork() system call is designed in such a way that it would create an exact copy of the parent process. Also we should wonder what it mean to have an exact copy of the parent process.

Let's discuss what it means to have an exact copy of the parent process. We will use the code example in Figure 2.1. In the code, after printing out "Hello!", the program calls `proc_id = fork()`. At the time the fork() system call is made, there will be a context switch from this program to the OS code that implements the fork() function. The details of this context switching are rather intricate; however, it is sufficient to note that P1 makes the fork() call, then the OS runs to execute the code for fork() routine that is implemented in the OS. Figure 2.2 shows a possible situation right before the OS started to run the fork routine.[2] It shows that P1 is the current process, but OS just started to run to execute the fork() routine in OS. P2 is just another program's process that is also loaded in the main program. Figure 2.3 shows right after switching to run OS from P1 but before running the fork() subroutine in OS. Notice that the CPU is currently running the OS. Finally, Figure 2.4 shows after executing the fork() subroutine in OS. Note that P1's child is created (its address space allocated, its PCB and other relevant data structures created), then it is inserted in the ready queue. As the OS finishes running the fork() subroutine, it will put the id value of the newly created child process to a designated register in the CPU so that it can be read by P1. This will allow P1 to get the id of the child process in its variable `proc_id` when P1 starts to run again.

Figure 2.5 shows the situation after the fork() subroutine is completed and P1 runs again. In line 7, it will get the child's process id (pid) in the variable `proc_id`. The parent then runs the `else` part of the program. The fork() system call is implemented in such a way that it will return zero to the child process. It is very important to understand which line of the code the child process will run as its first line of the code for the first time it is scheduled on the CPU. The simple answer to that is "from line 7" of the code. This is because when the parent calls fork() in line 7, the program counter value of the parent process is, surprise, line 7!! Because the fork() system call makes an exact copy of the parent to create the child process, the program counter value of the parent is also copied to the program counter

[2]The example scenario is intentionally simple and a bit fabricated. In real systems, there would be many more processes in the ready-queue and the situations would be more complicated. In this book, we will often use simple scenarios like this to focus on the concepts described.

of the newly created child process.

Figure 2.6 shows the situation when the child process runs. It will then run the `if` part of the program because the fork() call will return zero.

Program P1.

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  int main (void) {
4.  int val = 0;
5.  int proc_id;
6.  printf("Hello!\n");
7.  proc_id = fork();
8.   if (proc_id == 0) {
9.    val++;
10.  printf("I am the child;  val = %d\n", val);
11.  }
12. else {
13.  val--;
14.  printf("I am the parent; the child's ID is %d
15.         and val is %d\n", proc_id, val);
16. }
17. printf("Goodbye! Value of val = %d\n", val);
18. }
```

Figure 2.1: An example of a fork() system call

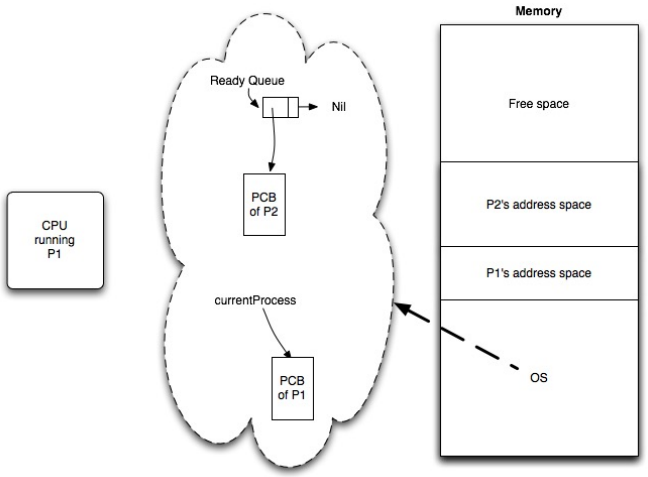


Figure 2.2: Just before calling fork() in Program P1

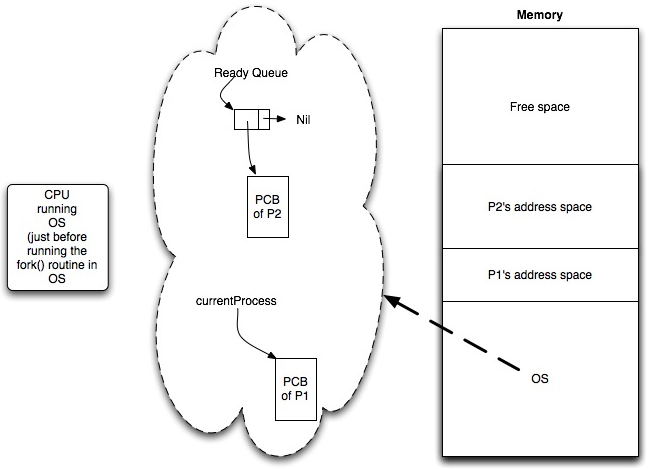


Figure 2.3: Right after calling fork() in Program P1 but just before executing the implementation of the fork() system call in OS. Currently the CPU is running the OS.

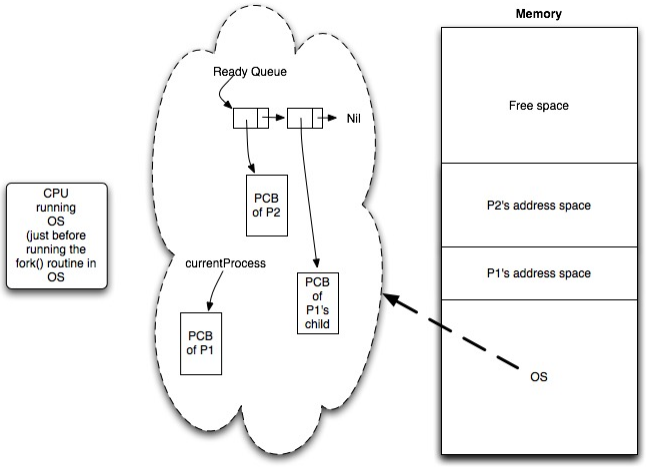


Figure 2.4: Right after finishing executing the fork() system call but before returning to executing P1

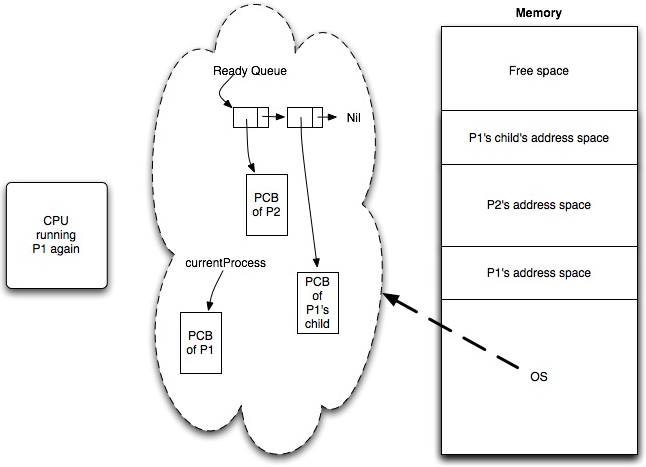


Figure 2.5: Right after finishing executing the fork() system call and returned to countinue to run P1. P1 gets the child's id (a non-negative value) as the return value of the fork() call in line 7.

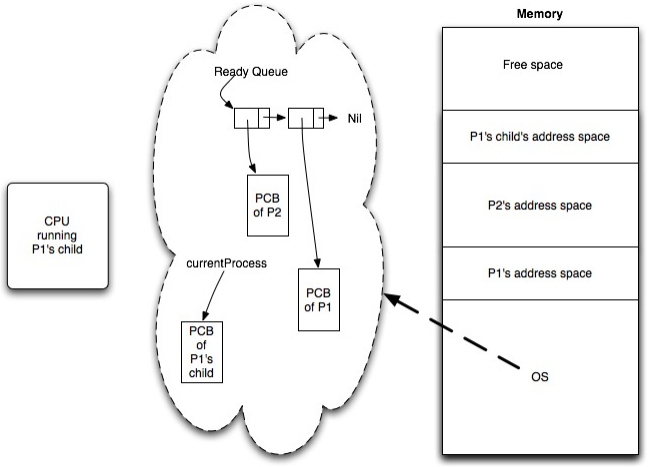


Figure 2.6: The scheduler picks the child process of P1 to run. The child process runs for the first time after it was created by the fork() call made by the parent. It will start running line 7, which will make the fork() call in line 7 again. Are you confused yet?

2.1.2 Implementation of `fork()` and related system calls

In the beginning of the book, I mentioned that in order to understand OS, we need to be able to comprehend the OS from various perspectives: the end user's, the application programmer's, the OS developer's, and the hardware designer's.

Figure 2.1 is the application programmer's view of `fork()`, which means that it shows how programmers would use the function in their application programs. Application programmers need to understand the input parameter requirements and what to expect on the behavior of the `fork()` call, including the return values for various scenarios. In other words, `fork()` does not have any input parameters and returns a new process id if it is called by a parent process, and returns 0 if it is called by the child process (i.e., the new process).

OS developers need to design and implement the `fork()` system call so that it will behave exactly in the same way as intended. Therefore, application programmers can use the function as intended and promised. In someway, this is the same as implementing a subroutine or a method function and using it in the program appropriately with the predetermined syntax. The difference here is that system calls are implemented by OS

programmers, while the application programmers simply use the system calls with the predetermined syntax.

Now let's discuss how the fork() system call function is implemented in Unix. Recall that a call to `fork()` will return 0 to the child process that is just created and the child id to the parent process. Roughly speaking, therefore, `fork()` needs to do three things: (1) create a child process that is identical to the parent; (2) return the child id if the call is made by a parent; (3) and return 0 if the call is made by the child. Notice I used "the" child not "a" child. This refer to the very child process that was just created by the parent process. Every process is a child process of some process, except the so-called "init" process, which is the very first process that is created during the boot process. So when we say "the child process" in the context of `fork()`, we mean the child process that just has been created and is to be scheduled to run on the CPU for the first time. Once it runs on the CPU, it will start executing a `fork()` call as the first instruction. Notice that, in fact, the child is not executing the `fork()` call from the beginning, it merely returns from it. This is because when the parent executes the `fork()` call, it runs in the kernel mode (or the kernel is running) so that it can create the child process and set up the return stack for child process and its PC any way we want. As the child returns from the `fork()` call, it will be a "potential parent," meaning that it can create its own child processes using `fork()` after returning.

```
1. Algorithm Fork()
2. Input: none
3. Output:
4.  to parent process, child PID number;
5.  to child process, 0
6. if (canBeParent) {
        // executing process is a parent process
7.  check for available kernel resources;
8.  get free process table slot and a unique ID
9.  make child state  being created.
10. copy meta-data of parent process to the child
11. copy parent's address space (text, data, stack, etc)
        to child's
12. do other initializations
13. change child state to  ready to run.
14. put the child to the ready-queue
15. return (child ID); }
16.else {
17. set this processs canBeParent = True;
18. and other necessary initializations
19. return (0) }
20.end Algorithm Fork()
```

Figure 2.7: A pseudo code for the Unix `fork()` system call

We now discuss an implementation of `fork()`. When a call to `fork()` is made, the OS will run the code in Figure 2.7. In the code, the first thing it checks is whether the call is made by a parent or by the child that was just created. If it is made by the child process, it will just make the `canBeParent` boolean variable to true and return 0 to the caller (i.e., to the child process). If the call is made by a parent process (i.e., its `canBeParent` is already true), then create a child process that is identical to the parent, insert the child process to the ready-queue, and return the child id to the parent (i.e., the calling process). In fact, the flag "CanBeParent" wouldn't be needed. In many actual implementations of `fork()` in Unix, the `fork()` call will manipulate the child's return stack and PC, so that when the child runs for the first time, it will start executing the "return from the fork() part" of the code. However, it is conceptually easy to think that `fork()` is called by the child as well, in my opinion.

Returning a value from an OS function to a user-level application can be

done through a CPU register. In other words, OS designer can designate a specific register to store the return value. The compiler creates a small code that puts the value in the CPU to the variable in the application program.

2.1.3 Using combinations of process management system calls

There are two other major system calls for process management in Unix. These are `exec()` and `waitpid()`. The `exec()` call has several variations in terms of input parameters. However, fundamentally they are similar. Try to run a command "`man exec`" in a Unix system and read the manual page. Other operating systems also have similar system calls.

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <unistd.h>
4. #include <sys/types.h>
5. main() {
6.   static char *argv[]={"ls","-a",NULL};
7.   execvp("ls", argv);
8. } // main
```

Figure 2.8: An example of an exec() system call

Figure 2.8 shows a simple example of `exec()` usage. The system call `execvp()` is a variant of `exec()`. An `exec()` call will replace the calling process's "core image." The core image of a process is basically the content of the address space and the contents of CPU registers (i.e., the CPU context). This means that after an exec() call is made, the calling process's program is changed to the core image of the program specified in the first parameter of the exec() call. In this example, this process's core image is replaced by the program `ls`. Thus, executing this process is the same as executing `ls -a`, which lists all files, including hidden files on the monitor screen.

Another process management system call is `waitpid()`. The system call `waitpid(pid, &status, 0)` makes the call wait (i.e., the processing calling `waitpid(pid, &status, 0)` blocked) in the `waitpid` queue until the process with the `pid` terminates. Note the `waitpid` queue is created and managed by OS, i.e., one of the OS's data structures. A process terminates in several different scenarios: normal exit, in which the process finishes executing its code normally; generating an offending error, such as division

by zero, can kill the process; and another process explicitly kills it. When a process terminates, an `exit()` system call is called. The `exit()` system call will release all resources held by the process exiting and clean up other necessary things. Even if the program does not make an exit() system call explicitly, the compiler puts exit() system calls to the appropriate places in the code.

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <unistd.h>
4. #include <sys/types.h>
5. main() {
6. pid_t  pid;
7. int status;
8. pid = fork();
9. if (pid != 0) { // parent code
10. waitpid(pid, &status, 0);
11. printf("******* I am the parent!!!\n"); }
12.else {
13. static char *argv[]={"ls","-a",NULL};
14. execvp("ls", argv); }
15.} // main
```

Figure 2.9: An example of using fork(), exec(), waitpid() and exit() system calls

Figure 2.9 shows how the four process management system calls work together. First, in line 8, a `fork()` call is made. The call creates a child process and returns the child's id to the parent. Then the parent process will continue to run the parent part of the program, which is `waitpid(pid, &status, 0)`. That will block the parent process in the waitpid queue until the child process with `pid` is done. If the `fork()` call in line 8 is called by the child process (recall the implementation of fork()), the child process will execute the `else` part of the program. The `else` part of the program will make a call to `execvp()`, which will replace the child process's core image with the core image of the program "ls." This will effectively make the child process an instance of "ls." When the child process is done, i.e., when the instance of "ls" is done displaying the list of files in the current directory, it will run line 15. In line 15, although not explicitly shown, an `exit()` call is made. In the `exit()` call, cleanup and release of resources (memory, file

pointers, etc.) are done. Then the parent process waiting on the waitpid queue will be set to ready to run and inserted back in the ready-queue, fulfilling the `waitpid(pid, &status, 0)` call.

We should notice that due to the nondeterminism in terms of the order of execution by the scheduler, the child process may finish running before the parent process even executes the `waitpid(pid, &status, 0)` call. If such a situation arises, the parent process won't be blocked in the waitpid queue when it executes the `waitpid(pid, &status, 0)` call; instead, it will simply go through the call without being blocked.

Think about how exec(), waitpid(), and exit() system call functions are implemented after understanding the semantics of these calls as explained above.

The `init` process, the mother of all processes

In Unix and Unix variant systems such as Linux, FreeBSD, etc., processes are organized in a tree structure. This means that except for one special process, `init`, all other processes have a parent process. The `init` process is created during the boot process as described in Chapter 1. All other processes are created using `fork()`.

When a computer system is turned on, it goes through the boot process described in Chapter 1. Let's assume the computer system that we are booting up is a server machine in a university. Students and faculty have computing accounts, and multiple of them can log in to the system remotely simultaneously. It may function as a web server as well.

As the system boots, after the init process has been created, the init process will use fork(), exec(), waitpid(), and exit() system call functions to create a remote login ssh session server process, a web server process, and other necessary processes. The remote login ssh session server process will listen to the ssh server port, which is usually port 22, for incoming login requests. If John attempts to log in from his home, the ssh session server will create a ssh session process (using a fork() call) for John's login attempt. This new session process is specifically designated to John's request, and it will run any permitted programs John requests after the authentication for login is successful. When John logs out, the session process calls exit(), and all resources allocated to the session process will be released and the process will be terminated.

In Unix and virtually many other OSs, there are processes that runs in the background (without being explicitly known to the end user). These processes, for the most part, serve the system's and user's needs. The ssh session server is one of these processes. Sometimes we call these processes "daemon" processes.

Except for the `init` process, all other processes have exactly one parent. When a process crashes due to programming errors, the child processes that belong to the process become "dangling nodes" in the process tree structure. These child processes without the parent process are called "orphan processes" in Unix. Unix assumes that child processes are created to help the parent process. Therefore, after the parent process is terminated, most likely the orphan processes are no longer needed, yet these processes still occupy memory and other resources in the system. A daemon that process that runs in the background once in a while finds these orphan processes and kills them to return the resources being occupied is called "orphan killer," which is a terrible term to use.

2.2 What are threads and why do we need threads?

We hear the term "threads" quite often. But the term "threads" was not so common in the 80s. I will explain the concept of threads in a historical account. To understand the concept, you must first clearly understand the concept of processes. I will review it now, but if you really are lost, I suggest you to go back to the process section and review the material.

A process is, by definition, a program in execution. Modern OS allows multiple processes to run concurrently. The point I want to make is that a process is an entity of its own, and it is *protected* from other processes. Consequently, a process should not access memory area of other processes. For example, in a personal computer, we can run a word processor and a web browser concurrently. In a server machines, multiple users can log in to the machine and run their processes concurrently, for example, one logged in user running a word processor and a voice call process and another user running a browser and picture viewer. It is clear that we don't want one process being able to access the memory area (the address space) of another process. If the OS lets a process access another process' address space, a voice call process would be able to access a word processes' address space. Or a clearer example is that one user's process would be able to access the address space of another user's process. This is a security breach.

In summary, a process is just designed to be its own entity. Think of it as an individual house. A house with resources and exactly one person living in it. Resources are all the appliances and furniture in it, and that person is only the entity that uses the resources.

A process traditionally had exactly one *thread of execution.*[3] This design has consequences. Consider a program that reads from a file stored in a hard disk. A web server process is one such a program. A web server awaits for incoming requests for web page contents stored in a hard disk, and sends the content to the remote requester.

Recall that the amount of time that takes to access a hard disk is "millennia" from the CPU's perspective. Therefore, a web server process with only one thread of execution would have to be blocked while the access to the hard disk is being made. This would allow other processes waiting in the ready-queue to run until the disk access is completed. See Figures from 1.30 to 1.33. It seems to be all fine up to this point. However, what happens if while the server process is being blocked, another request from a remote

[3]Think about the person who lives in the house above. She is the person who uses all the resources within the house to perform tasks.

user arrives? Note that the server process has no control over how many requests should come in and when. Can the Google search engine tell which remote users can request web pages and when? The answer is no. Of course the simplest way is to buffer the incoming requests in a queue and serve each request one by one. But this means that the server's performance is dependent on the hard disk access time, which is really slow.

Another problem of having only one thread of execution is that context switching between two processes takes a long time. We discussed the round-robin scheduling of processes in the last chapter. In the scheduling scheme, each process is given a fixed amount of time, called quantum, usually in some milliseconds, to run. However, if the current process requests a blocking system call, the process must be blocked until the system call is served. A typical system call that blocks the calling process is an I/O request. This means that the amount of quantum allocated to the process is not used entirely. A process may have something else to do while waiting for the I/O to be completed. Consider a web server program again. A pseudo code for a single threaded web-server is given in Figure 2.10. The server process listens to the specific incoming communication port and sends the requested web content to the remote user. If the requested web content is not in the cache memory, it will have to be retrieved from the hard disk. A hard disk read is scheduled and the server process gets blocked until the hard disk read is completed.

```
Webserver() {
  While (true) {
  request = listen_to_port();
  if (request_is_in_cache())
   return request.content;
  else
   {
   make_disk_read(request); // this process blocks here
   return request.content;
   }}}
```

Figure 2.10: A single threaded web server

```
Webserver() {
  While (true) {
  request = listen_to_port();
  Create_Thread(request);
  }
```

Figure 2.11: A multi-threaded web server

On the other hand, in the multi-threaded web-server example in Figure 2.11, the parent server will just listen to the requests and create a worker thread for each request. Each worker thread will perform actual work, such as checking the cache and making a hard disk request, if necessary. This way, the main thread is not blocked, and it can keep serving incoming request.

Notice that a context-switching between a thread to another thread is much faster (because not much of the data are to be saved and restored compared to a switch between processes).

In our analogy of house and inhabitants in the house above, a multi-threaded process means that more than one person lives in a house, and all can share resources in the house freely. In the web server example, the cache memory and hard disk content can be shared resources. Most people would agree that sharing resources in another house with people living in the other house is more di?cult than sharing within a house.

Blocking calls and non-blocking calls

- Blocking system calls: A blocking system call is a system call that blocks the process that calls it. I/O commands, such as read and write system calls, are typical blocking system calls. When currently running process calls a read call, for example, the process that called it is blocked and therefore gives up the CPU until the read call is completed. This is explained in Figures from 1.30 to 1.33.
- Non-blocking system calls: A non-blocking system call is a system call that does not block the process that calls it. Therefore, even as the OS serves the call, the process that called it does not get blocked. This means that the process will continue to run as the call is being served by the CPU.

2.2.1 Data structures of process and thread

Processes and threads are created and managed by OS. We learned a minimal PCB data structure type for processes in Chapter 1. See Figure 1.18. Figure 2.12 shows a typical thread structure. One main thing that is missing from the process PCB is the pointer to the address space. It is safe to understand that there is a process that contains a thread. In other words, when a process is created, it automatically has one thread of execution. A thread of execution requires a program counter variable and other necessary variables. Within a process, multiple threads can be created.

```
typedef thread_structure {
 char* thread_name;
 int thread_number;
 int thread_status;   // Ready, Run, and Blocked
 int program_counter;
 int stack_pointer;
 int registers [32];
}
```

Figure 2.12: A minimal thread structure

2.3 Typical depiction of process and threads

Figure 2.13 shows a typical depiction of threads.

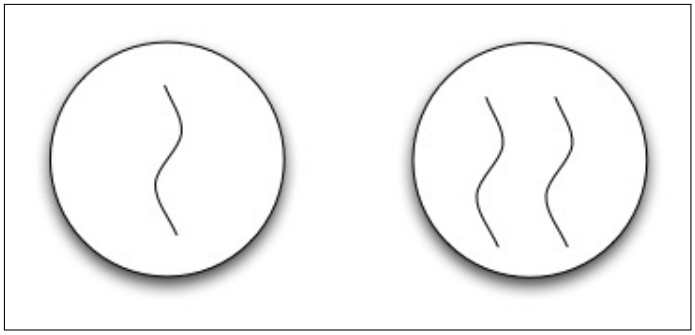


Figure 2.13: Processes are represented by circles, and threads are represented by wavy lines, in general. A process must have at least one thread of execution, i.e., one thread. As needed, additional threads can be created.

2.4 Examples of thread systems

By now, we understand that one or more threads exist within a process. There must be at least one thread in a process because "thread" means "thread of execution." We now talk about some existing thread systems we encounter commonly.

2.4.1 Java Virtual Machine

A program written in Java programming language is compiled by a Java program compiler. A Java program compiler takes a program written in Java and generates a logically equivalent program in *Java bytecode*. In order to run a Java bytecode program, a Java Virtual Machine (JVM) must be installed on the computer system. In a pure form, a JVM is a process running on a native OS. For example, if a JVM is running on a Linux machine, it is a Linux process. Theoretically speaking, you should be able to run as many JVM instances as you want, as long as there are enough computing resources, such as memory, etc. So each JVM instance is a process running on, say a Linux machine. When you run a Java program on JVM, that program is a thread from the Linux machine's point of view.

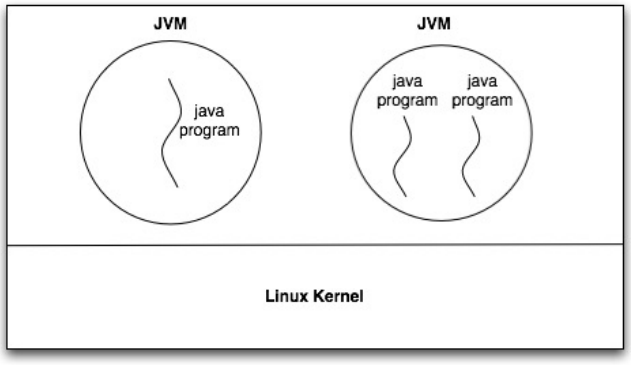


Figure 2.14: Two JVM processes running on Linux. Within one JVM, one Java program is running, and in the other JVM, two Java programs are running.

In theory, JVMs should not be any different from any other processes. Therefore, a JVM process or a web browser process should be treated the same by the host OS. However, because Java programs are ubiquitous, for

various reasons, host OS treat JVM differently. There is no theoretical reason why a JVM process should be different, though.

If threads within a process are visible (i.e., known) to the host OS, we generally call them *kernel-level threads*, which means the host OS (the kernel in a broad sense), knows the existence of them and therefore can manage them—such as in scheduling.

If threads within a process are completely hidden from the host OS, we generally call them *user-level threads*; this means that these threads are not known to the OS. Therefore, in general the host OS cannot manage them.

Note that for a single CPU (with single core) system, there is only one thread running at any given time. Context-switching can happen between threads within a process or between a thread in one process and a thread in another process. When threads are implemented at the kernel level, scheduling can switch from a thread in one process to a thread in another thread. If threads are implemented in the user level, because the kernel does not know the threads exist, such a scheduling is not possible.

In Java programming language, there are several ways to create threads. Figure 2.15 shows a Java program that creates two thread. The HelloThread class is defined to be a thread class, the class that defines threads in Java. Then whenever an object HelloThread is instantiated from the HelloThread class, a thread object is created. In line 5 and 6, two threads are created in turn. In line 7 and 8, the threads are started. As in processes, when threads are started, they are inserted in the ready queue so the scheduler can run each in turn when appropriate.

2.4.2 Linux threads

Linux implements portable operating system interface (POSIX) thread library. The POSIX system is a standard from IEEE Computer Society. Programmers can use the thread implementation, that is, to create and use threads to develop multi-threaded application programs. C language is a good language to develop multi-threaded application on Linux. POSIX threads are shortened to "pthread." Figure 2.16 shows a very simple example for creating and running POSIX threads in C. The program is equivalent to the Java version shown above. In order to use the pthread library, the programmer must include the pthread.h header file so that the compiler would know where it would find the pthread library routines during the compilation and linking phases. The type `pthread` is used to define variables with thread types. The thread library function `pthread_create` is used to create threads. The first argument of the function takes the thread structure (i.e.,

```
1. import java.io.*;
2. import java.lang.*;
3. class MyThreadExample {
4. public static void main(String[] args) {
5.   HelloThread ht1 = new HelloThread(1);
6.   HelloThread ht2 = new HelloThread(2);
7.   ht1.start();
8.   ht2.start();
9.   }
10. }
11. class HelloThread extends Thread {
12.   int threadID;
13.   HelloThread(int threadID) {
14.     this.threadID = threadID;
15.    }
16.   public void run() {
17.     System.out.println("Hello from thread " +
18.          this.threadID);
19.    }
20. } // end Thread
```

Figure 2.15: An Example Java program creating threads and running them.

variable), the second argument is left to NULL for this example. Usually, it is used to specify thread attributes, which is beyond the scope of this book. The third argument is the function associated with the thread being created. That is, the subroutine program that will be executed when the thread runs. The last argument is the parameter for the subroutine. The subroutine to run in this example is `PrintHello`. I encourage the readers to compile the program and run it on a Linux or a Unix-like system. Two threads within a process, in this case `HelloThread1` and `HelloThread2`, share the memory area and the address space. That's why both threads can access the same function `PrintHello`. It is not possible or at least not easy for two processes sharing such a function.

It is important to understand that there are in total three threads in this example. Some may think there are only two. Remember that there is at least one thread of execution for a process. When this main program starts to run, the so called the main thread is running. As the computer runs line 11, 12, and 13, two more threads are created. From line 10 to 16 are the thread of execution for the main thread.

We now explain the thread library function `pthread_join`. It is important to understand that the `pthread_join(HelloThread1, NULL)` function will stop the main thread until `HelloThread1` is done.

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <pthread.h>
4. void *PrintHello(void *threadId) {
5.  printf("Hello from thread %ld\n", (long) threadId);
6.  pthread_exit(NULL);
7. }
8. int main() {
9. pthread_t HelloThread1, HelloThread2;
10. pthread_create(&HelloThread1, NULL,
                            PrintHello, (void*) 1);
11. pthread_create(&HelloThread2, NULL,
                            PrintHello, (void*) 2);
12. pthread_join(HelloThread1, NULL);
13. pthread_join(HelloThread2, NULL);
14. }
```

Figure 2.16: An Example POSIX thread program to create threads and run them

Figure 2.17 shows an illustration of the program flow of the pthread example in Figure 2.16. The main thread first starts then creates two threads. These two threads will use their CPU time at the will of the scheduler. There is no specific order in which thread would run first. The main thread however will wait until `HelloThread1` finishes then wait for `HelloThread2` to finish. It is possible that by the time the main thread executes a join function, the corresponding child thread may have already finished. If for example, `HelloThread1` is not done when the main thread runs `pthread_join (HelloThread1, NULL)`, the main thread will be blocked (i.e., wait for HelloThread1 to finish.) If HelloThread1 is already done by that time, the main thread won't be blocked and will continue on the next line. The same situation happens for `HelloThread2`.

We have already seen through the join function that there is a way to synchronize thread execution. We can do the same among processes using the `waitpid()` function.

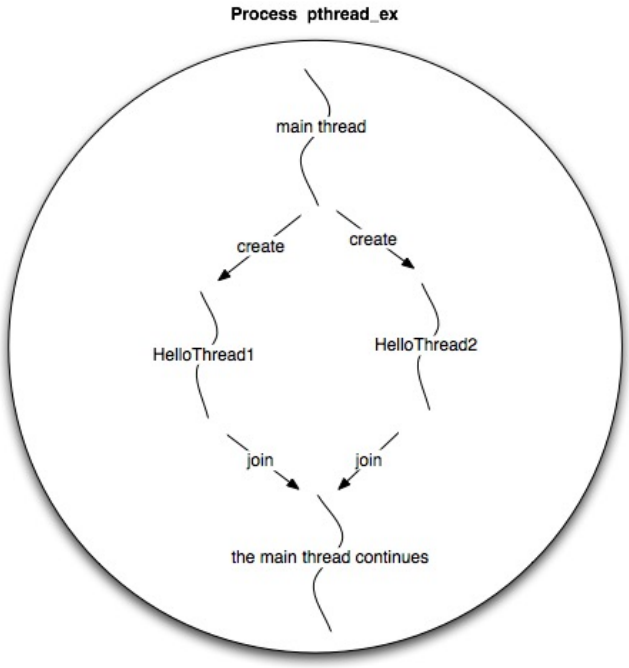


Figure 2.17: The depiction of the pthread example in Figure 2.16. The main thread starts first and creates two threads. It waits until both threads are done and continue.

2.5 Schedulers, scheduling algorithms, timer interrupts, and I/O interrupts

A scheduler is a subroutine (or a class object in the object oriented programming) in OS. There are several different schedulers in an OS, but when we simply say "scheduler" without a modifier, we mean it to be a CPU scheduler. In a single CPU computer system, there is usually only one CPU scheduler. So we will call it *the* scheduler from now on by default unless we explicitly mention otherwise.

So what is the scheduler for? When we run multiple programs concurrently, we have exactly one program that is physically executed by the CPU at any given time. Other processes that are ready to run must be queued up in the ready-queue. The scheduler will pick a process from the ready-queue when the currently running process cannot continue to run on the CPU for various reasons. Some of the reasons are already discussed in the

previous chapter. One reason is that the current process makes a blocking system call, such as an I/O. When such a situation occurs, the CPU will start running a part of OS code to do all the necessary work for the I/O (which we discussed in the previous chapter), then the OS must choose the next application process to run. To choose which process in the ready-queue should run next, the scheduler subroutine is called. The scheduler subroutine will find an appropriate process from the ready-queue according to the scheduling policy and *run* the selected process. By *running* the process, what I mean is that the scheduler will load the program counter value of the selected process from its PCB to the program counter register of the CPU. Remember that to run any program, we must load the program's program counter value, which is pointing to the next instruction to be executed, to the program counter register in the CPU. This is called context-switching.

Context-Switching Steps

A context-switching occurs when the currently running process gives up the CPU (either voluntarily or forcibly) and another process starts to run. Users cannot perceive this behavior of computer systems because it occurs so fast. In fact, because of this speed of switching from one process to another, humans have the illusion of running multiple programs at the same time even if there is only on CPU in the computer system.
Fast as it may be, a context-switching occurs in many steps. In a larger view, a context-switching involves three steps: stopping the current process, starting to execute the relevant OS code, and then starting the next process. More detailed steps are explained below.

1. Currently running process, say P1, gives up the CPU and CPU starts to run the relevant OS code.
2. In the OS, the content of CPU registers that belong to the previously running process, P1, are saved to P1's PCB. Note: in order to make OS start to run in Step 1 without overwriting much of the CPU content for P1, in Step 1, hardware will save the PC value of P1 to some known place. So in Step 2, it can be copied to P1's PCB.
3. OS does other necessary things for P1. For example, if P1 makes an I/O call, it will perform an I/O request and put P1 to an I/O waiting-queue. P1 is now in the blocked state if it just made an I/O call before being stopped. In another scenario, P1 could be inserted to the back of the ready-queue (in the round-robin scheduling as explained below).
4. OS now calls the scheduler; the scheduler will choose the next process, say P2, to run from the ready-queue according to the scheduling policy.
5. The register values of P2 are copied to the corresponding physical registers in the CPU, including the PC value to the physical program counter register to start P2.

2.5.1 Round-robin scheduling

Another reason that the currently running process gives up the CPU that its *quantum* has expired. There are many different scheduling policies. In the round-robin scheduling policy, the quantum is the amount of CPU time allocated to a process during which it can continuously run unless it voluntarily gives up the CPU.

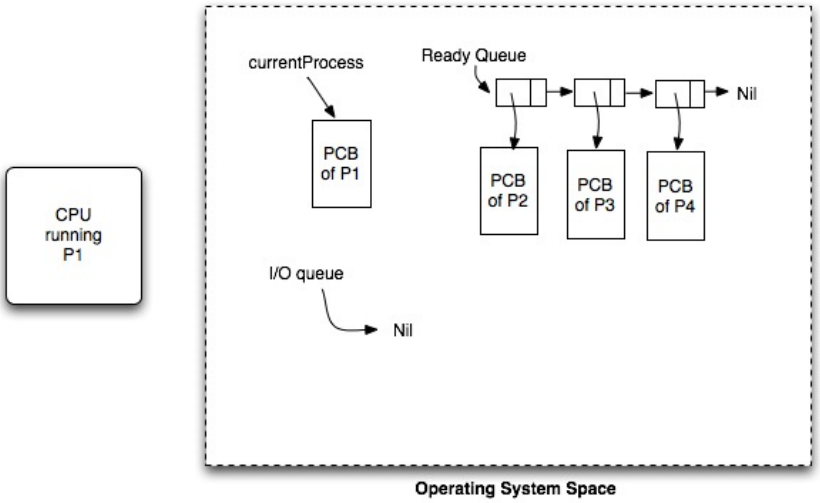


Figure 2.18: At the beginning of round-robin scheduling. P1 is currently running, and all other processes are waiting the ready-queue.

For example, in Figure 2.18, process P1 is currently running, and all other processes are waiting in the ready-queue. Let's say that the quantum size is $3ms$. This means that once a process is scheduled to run on the CPU, it will run on the CPU for $3ms$ continuously unless it voluntarily gives up the CPU (for example, making a system call or terminating) or some other higher priority interrupt happens. Once the $3ms$ period is over, the next process in the ready-queue runs. Let's say that P1 runs its $3ms$ quantum without any interruption. After the $3ms$, a timer interrupt occurs to let the CPU know that the quantum expired. Recall how an I/O interrupt was handled in the previous chapter. We also discussed timer and timer interrupts. The steps of the timer interrupt service is similar to I/O interrupts. After P2 started to run, now the situation looks like in Figure 2.19.

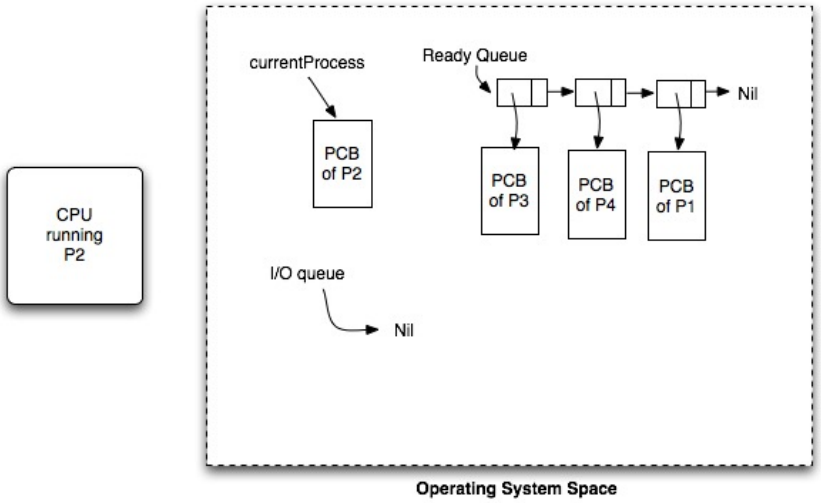


Figure 2.19: Now P2 runs after P1 has used up all of its quantum.

Let's say for the sake of explanation, P2 runs for 2ms and then makes an I/O system call–read(). Since P2 cannot continue to run, even if it still has $1ms$ of the $3ms$ quantum, it must be switched out from the CPU. The scheduler subroutine in OS runs and it chooses the next process to run, which is P3.

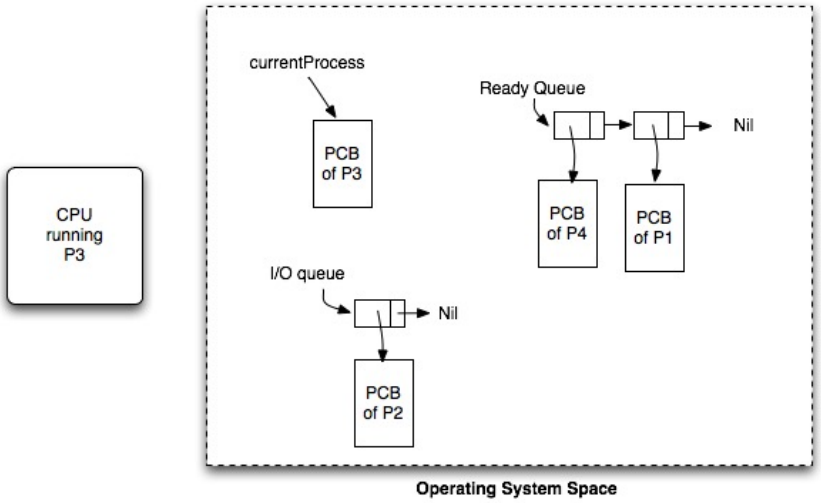


Figure 2.20: P2 gets blocked in the I/O queue after making a read() system call until the I/O is completed. Meanwhile, P3 starts to run.

Note that context-switching still takes time. Let's say it takes $1ms$. So in our example, if we start counting from at time 0, P1 runs from 0 to $3ms$, then $1ms$ for context-switching, and P2 runs from at the *4th ms* and context-switched out at the *6th ms* due to the read() system call.

2.5.2 Other scheduling policies

Another scheduling policy is first-in-first-out (FIFO). In its purest form, a FIFO scheduling will work like the batch processing we discussed previously. It will run programs one by one in the order of introduction, without switching to the next one. It will finish the current process before starting a new one. However, if we think about it, the round-robin scheduling is also somehow FIFO. It will also run processes one by one in sequence although processes are interrupted in the middle.

Priority-based scheduling policy chooses the next process to run based on the priority of each process waiting in the ready-queue. The scheduler will parse through the ready-queue or sort the ready-queue in the order of the priority and choose the process that has the highest priority. In fact, all scheduling policies are in a way a priority-based scheduling policy. FIFO gives the priority to the ones that came first. Round-robin more or less tries to give priority to the processes that havent been scheduled for a long time.

Computer systems exhibit dynamic behavior. Processes are introduced (loaded) to the system all the time (think a server system). If higher-priority processes are introduced while lower-priority processes are waiting in the ready-queue, a naive priority-based scheme will keep executing higher priority processes. Consequently, lower-priority processes will forever wait in the ready-queue, never being run on the CPU. We call this phenomena *starvation*.

Multi-level priority-based scheduling attempts to resolve this issue. Instead of having just one ready-queue, multi-level priority-based scheduling maintains multiple queues. For example, there can be five ready-queues, each having its own priority: the level one queue has the highest priority, and the level five queue has the lowest priority. When processes are introduced, they are inserted to the lowest-priority queue or to the queue that is associated with their initial priority values. The scheduler will pick one, maybe the highest priority process among the ones in the level one queue, for the next process to run. If a process in a lower-level queue has been waiting in the queue for a long time, the process will move up to the next higher-priority queue after some time. We call this aging. This way, eventually all processes will move up to the highest-priority queue and have chances to run. This resolves the starvation issue.

Preemptive and Non-Preemptive Scheduling

- A preemptive scheduling allows the current process to be stopped and switched out for the next process forcibly (preemptively). Notice that this context-switching is not the "will" of the current process. The round-robin scheduling is a preemptive scheduling, because a timer interrupt causes the current process (or thread) to be switched out.
- A non-preemptive scheduling allows the current process to run as long as it wants until it voluntarily gives up the CPU. Some of the events for the current process to give up the CPU voluntarily would be making an I/O call or finishing itself (i.e., calling an exit() call). A non-preemptive FIFO scheduling is a non-preemptive scheduling with the first-in first-out priority order of execution.

Note that if the current process (or thread) causes a trap, an error state, CPU will stop its execution and run OS to take care of the situation, no matter whether the OS is using a preemptive or a non-preemptive scheduling policy.

2.5.3 Scheduling in batch systems, real-time systems, and other special systems

CPU scheduling ultimately decides the next process to run among the available ready-to-run processes. There are a lot of different ways of choosing the next process depending on the goals of the OS designer. Batch systems would have different kinds of scheduling policies, for example, from interactive systems. In real-time systems each process has a deadline to complete. The processes must finish within their deadlines; otherwise something bad can happen. In hard real-time systems, this bad thing can be catastrophic. For example, an auto-pilot system is a hard real-time system. If opening of landing wheels is not done in a timely manner, the plane can have an extremely dangerous landing.

Soft real-time systems may have a lesser degree of seriousness when they fail to honor the deadlines of processes. Multi-media systems are soft real-time systems. If rendering of movie frames is not done in a timely manner, user experience will suffer, but no one will be physically hurt. Because of the seriousness of consequences of failing to meet deadlines in hard real-time systems, we don't "play" with scheduling in such systems. Instead, we have redundant systems so that if one system fails or is overloaded, another system can execute the processes in a timely manner. Soft real-time systems, however, can withstand some bad performances. Therefore, there are many innovative scheduling algorithms for soft real-time systems.

2.6 Benefit of using multiple threads in programming

What are the benefits of using multiple threads in programs? One benefit is parallelism. We can run multiple threads within one process. Why is running multiple threads within a process a good thing? To understand this, recall in round-robin scheduling, each process is given a quantum to run on the CPU until it is interrupted by the timer to run the next process in the ready-queue. If there is an I/O system call while the current process is running, say at half way through its quantum, the process without multiple threads in it will have to be blocked and let the next process run even before the quantum expires. If there are multiple threads in the process, the thread that made the I/O call will be blocked but most likely not all threads in the process. It is highly likely that at least one thread within the process could be ready to run at the time the thread made the I/O call;

we can switch to the ready thread when we block the thread that made the I/O call. Remember that switching from one process to another process is more expensive (time consuming) than switching from one thread to another thread within the same process. Therefore, having multiple threads within a process will reduce the chances of process switchings; consequently more time is used to run application programs rather than OS code (switching is done running OS code).

Another benefit is that sometimes thinking in terms of multiple threads is more natural for the problem. In our example of a web server, having to spawn off a worker thread for each request is conceptually simple and efficient. If you are developing a program that controls a robot with sensors and an ability to fly, creating designated threads for sensor I/Os and controlling the flight would be conceptually natural. An alternative is to take turns between sensing and flight control with one process.

2.7 Wrapping up

We summarize important concepts for this chapter.

- The differences between process and thread
- How `fork()` can be used to create a new process
- How `fork()` is implemented in Unix
- How other system calls work and are implemented
- Why threads are useful; why using processes alone is enough.
- How different scheduling algorithms work.

As always, when you learn a new concept, you should think about how to implement the concept in OS and which part of OS should be modified.

Exercises

Ex. 2.1 — We discussed three cases in Figure 2.1. Explain how all these cases are fundamentally the same. You may use examples.

Ex. 2.2 — Study Z80 microprocessor. Explain each general purpose register and special purpose register.

Ex. 2.3 — On Page 56, we discussed three different ways a process can be created. The third case, in which a currently running process creates a new process, in general covers the first two cases. Explain why.

Ex. 2.4 — In the example of `fork()` system call in Figure 2.1, which line is executed as the very first line by the child process and why?

Ex. 2.5 — Study how calling and returning from a subroutine in a program written in C (or most procedural languages) and compare how this technique is related to calling a system call and returning from it. Hint: you can search for terms like "compilers," "activation record," "calling and returning from a subroutine." This is a hard question but one with a substantial benefit.

Ex. 2.6 — Write a small program that uses `fork()` and `exec()` to run the `ls` command.

Ex. 2.7 — Every program ends with calling an `exit()` call. However, programmers usually forget to put an `exit()` call at the end of programs. How does a call to an `exit()` call happen if the programmer forgets to put the call?

Ex. 2.8 — The `exec()` call replaces the calling process' core image. Explain this "core" precisely and the effect of the replacement.

Ex. 2.9 — The amount of time that takes to make a context-switch between threads versus that of between processes is much shorter. Explain why.

Ex. 2.10 — We discussed a web server example of using threads. Discuss another application example that can benefit from using multiple threads. Justify.

Ex. 2.11 — Find an example of non-blocking system call. Explain.

Ex. 2.12 — Figure 2.12 defines minimal data that are needed for a thread.

Explain whether we can have a thread with the program counter value. Is it possible or not? Explain.

Ex. 2.13 — Figure 2.12 defines the minimal data that are needed for a thread. What if we don't have stack pointer value in the thread structure? Can we still have a thread system? If so, what functionality would be limited?

Ex. 2.14 — Is the JVM implemented in the user space or kernel space?

Ex. 2.15 — In a multi-threaded system with a round-robin scheduling at the process-level, implement a thread-level round-robin scheduler in pseudo-code.

Ex. 2.16 — Discuss pros and cons of implementing a thread system at the kernel level or at the user level. Is implementing a thread system at the user level useful anyway, since the kernel can only see I/Os are done at the process level?

Ex. 2.17 — Write a pseudo-code for `exec()`, `fork()`, `waidpid()`, and `exit()` system calls. Explain how these system calls are inter-related.

Ex. 2.18 — In Unix, there is a global table called the process table. Explain what this table is for and how it is being used.

Ex. 2.19 — In Unix, there is a category of processes called *zombie* process. What are these?

Ex. 2.20 — What is the UNIX command to see a list of all processes running in the system currently?

Ex. 2.21 — In multi-level priority scheduling, can a process wait in a queue forever? How about in the shortest job first scheduling?

Bibliography

[1] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Comput. Surv.*, 15(1):3–43, March 1983.

[2] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM.

[3] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[4] M. Ben-Ari. *Principles of Concurrent Programming*. Prentice Hall Professional Technical Reference, 1982.

[5] C. W. Bettcher. Thread standardization and relative cost. *j-COMP-ARCH-NEWS*, 2(1):9–9, January 1973.

[6] R. B. Bunt. Scheduling techniques for operating systems. *Computer*, 9(10):10–17, October 1976.

[7] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[8] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, June 1971.

[9] Yoav Etsion, Dan Tsafrir, and Dror G. Feitelson. Process prioritization using output production: scheduling for multimedia. *ACM Trans. on Multimedia Comput. Commun. and Appl. (TOMCCAP*, 2:318–342, 2006.

[10] Per Brinch Hansen. Short-term scheduling in multiprogramming systems. *SIGOPS Oper. Syst. Rev.*, 6(1/2):101–105, June 1972.

[11] Per Brinch Hansen. *Operating System Principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1973.

[12] Carl Hauser, Christian Jacobi, Marvin Theimer, Brent Welch, and Mark Weiser. Using threads in interactive systems: A case study. *SIGOPS Oper. Syst. Rev.*, 27(5):94–105, December 1993.

[13] Butler W. Lampson. A scheduling philosophy for multiprocessing systems. *Commun. ACM*, 11(5):347–360, May 1968.

[14] Phillip A. Laplante, Eileen P. Rose, and Maria Gracia-Watson. An historical survey of early real-time computing developments in the u.s. *Real-Time Systems*, 8(2):199–213, 1995.

[15] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[16] Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[17] Paul R. McJones and Garret Frederick Swart. *Evolving the UNIX system interface to support multithreaded programs: The Topaz Operating System programmer's manual*, volume 21 of *Systems Research Center*. Digital Systems Research Center, Palo Alto, CA, USA, September 1987.

[18] Toshimi Minoura. Deadlock avoidance revisited. *J. ACM*, 29(4):1023–1048, October 1982.

[19] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[20] Z. Rosberg and I. Adiri. Multilevel queues with extremal priorities. *Journal of ACM*, 23(4):680–690, October 1976.

[21] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[22] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. *SIGOPS Oper. Syst. Rev.*, 23(5):159–166, November 1989.

3

Memory Management

Why do we need memory management? Roughly speaking, there are two reasons for memory management. The first is *protection.* For example, in a mono-programming system, there are two programs loaded in the main memory: the OS and the user program, as shown in Figure 3.1. The memory management scheme must protect the OS from the user program and vice versa. In reality, protection is needed more for the OS to prevent the user application program from accessing the memory area that belongs to the OS. In this example, it is the address from 0 to (0x80-1). If this protection is not ensured, users can write a program that can access the OS area and run the program as if it is a part of OS and damage the computer (or data stored). Many computer hacking methods involve so-called *buffer overflow methods*. A buffer overow method can change the next instruction to be executed to a jump instruction that will jump to a part of OS that has a special privilege. Once the hacker gets hold of one of the OS subroutines, she or he can run subroutines that are only meant for system administrators or the OS itself.

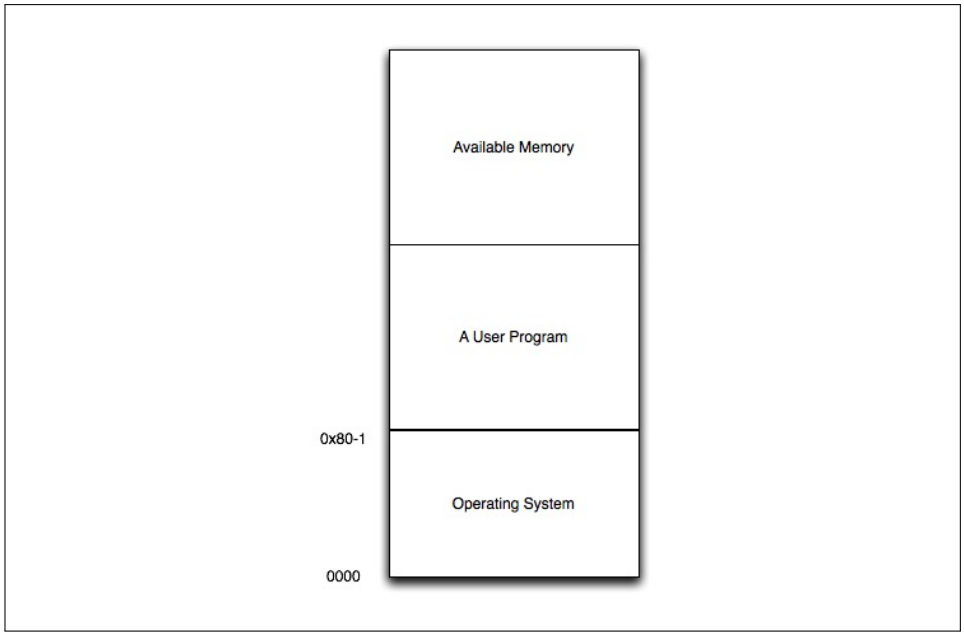


Figure 3.1: A system that allows only one user program to run at a time. In this case, we have only two programs loaded in the main memory at most: the OS and the user program. Some traditional batch systems or earlier personal computers are similar to this.

Therefore, we must always check whether the next program counter value is within the legal range of the currently running program. Which component in the computer system would do this? It is being performed by the hardware that is a part of the Memory Management Unit (MMU). The MMU will check if the PC value is within the permitted range of the currently running process. In our example, when the OS is running, the MMU will check if the PC is within the range of 0 to 0x80-1. When the user program is running, the MMU will check if the PC value is within 0x80 to 0x104-1. This comparison can be done in a very simple way, using some logic gates. Figure 3.2 shows this idea. Notice that the CPU always assumes that programs are loaded in terms of a virtual address space. That is, a program always starts from address 0, and it is loaded in a contiguous area of the main memory.[1] In multi-programming, protection is also needed from one user program to another.

[1]Note that we have not discussed non-contiguous memory allocation yet. In other

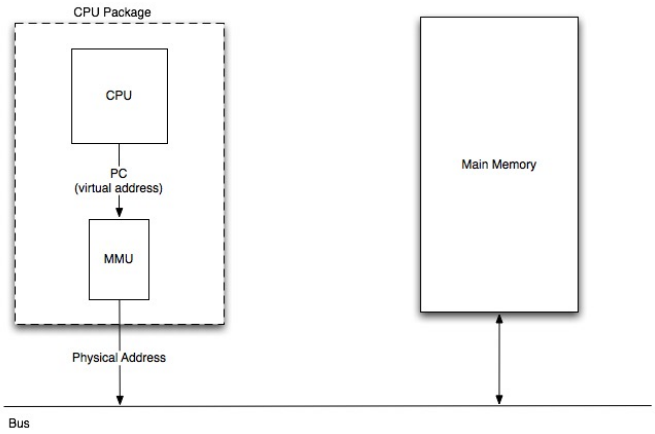


Figure 3.2: A simple MMU for contiguous memory management.

The second reason for memory management is for the utilization of memory. OS attempts to maximize memory utilization so that more programs are loaded at the same time and run concurrently. Sometimes we want to run programs that are larger than the amount of memory available. Memory-management schemes such as *overlays* and *virtual memory* make this possible. Most modern desktop computers have quite a large amount of memory compared to old computers. For example, NASA's original Voyager has two processors with each having only 4K memory. At the time of writing this book, 16GB of main-memory size is common in desktop computers. The point that I am making is that no matter how much memory computers have, programmers will always want more. Therefore, efficient memory management is needed.

---

words, a contiguous memory-allocation scheme does not break a program into several pieces and load the pieces at different parts of main memory. We will discuss more details of this method later. It is important though to understand how the contiguous memory allocation works.

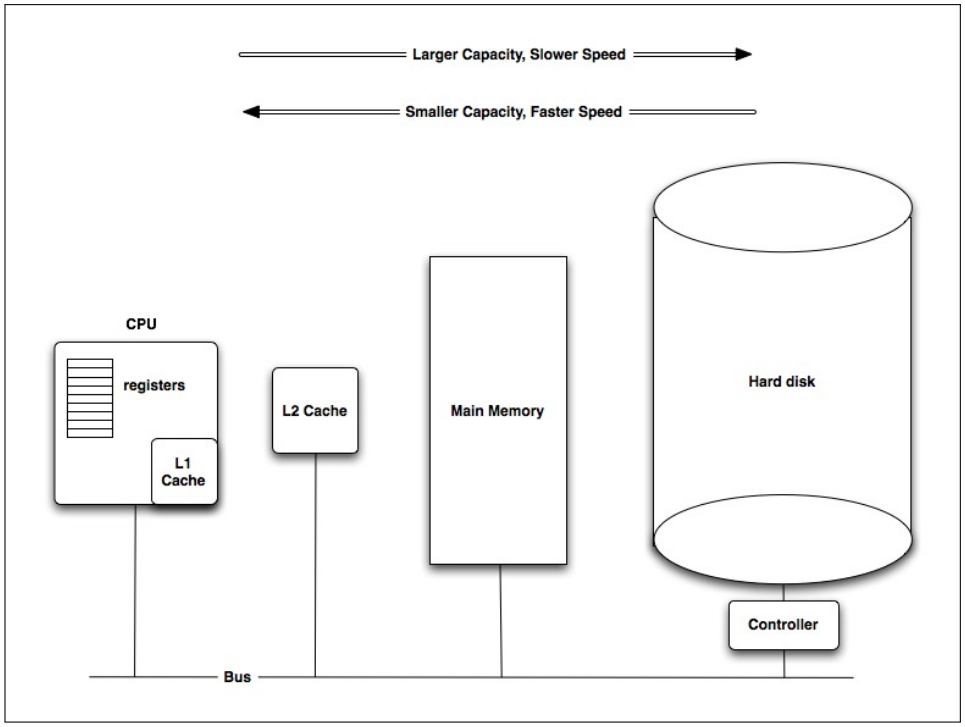


Figure 3.3: The memory hierarchy

The third reason for memory management is to support fast instructions and data accesses. Instructions (i.e., programs) and data are stored in memory during program execution. CPU performs computations, and computations require instructions and data. In order to understand how memory management can support fast instruction and data accesses, we need to review the concept of the memory hierarchy. As in Figure 3.3, the memory devices that have the smallest capacity are registers. Registers are in CPU (and in device controllers). A CPU generally contains general purpose registers and some special purpose registers such as the program counter. The general purpose registers are used to store temporary results of computations. For example, `ADD R0, R1, R2` instruction performs `R2 = R0+R1`. Many CPUs have 32 general purpose registers. Each register is usually one word. One word is usually 4 bytes. Next in the the memory hierarchy is L1 cache memory. It has a larger capacity than the CPU registers combined. Typically Intel CPUs have about 32KB for data and instruction cache. The L1 cache is usually housed within the same package as the CPU.

It is closer to the ALU and other CPU logic circuits so that access time can be faster. The next in the hierarchy is the L2 cache.

A L2 cache size can average 2MB but it is slower to access than L1. The speed of a memory device generally depends on two factors: the device's hardware characteristics, such as refresh rates, and the size. The hardware characteristics can be improved through better design and materials; however, size is a different matter. When the size grows the control logic circuit size grows, and covers the large number of memory elements in the memory device. Also, accessing the exact element out of a large number of elements takes a long time. Just think about finding specific information on one page of paper compared to a large volume of papers. After L2, we have the main memory. The main memory can be easily several GB or even TB. As we just discussed, the access time would be even slower than that of L2. Memory access time is measured in the tens of nano seconds range. Traditionally, the next in the hierarchy is the hard disk drive. Hard disk drives are huge and their sizes range from hundreds of GB to even Petabytes. The access speed of hard disk drives are extremely slow compared to the devices we discussed so far. The speed was over 10 milliseconds when this book was written. Notice that all other devices except the hard disk have an access time in the nanosecond ranges. A millisecond is a million times slower than a nanosecond. The main reason for hard disks being so slow is that they are electro-mechanical devices. The mechanical part of the hard disks significantly slows down the device access time. More recently, many computers come with Solid State Drives (SSD) instead of hard drives. SSD are faster than HDD but they are more expensive. Therefore, the capacity of SSD tends to be smaller in computers.

3.1 Memory Hierarchy and The Principle of Locality

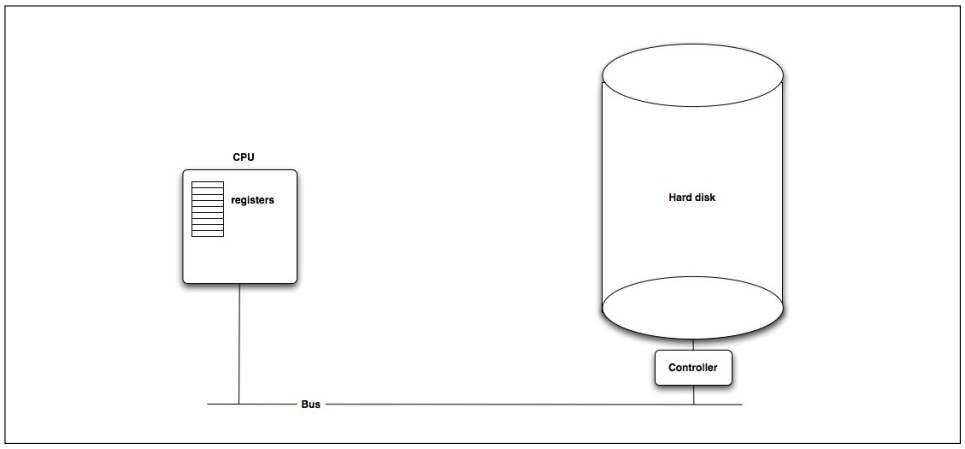


Figure 3.4: An imaginary computer with only hard disk; there are no other memory devices in the memory hierarchy

Imagine a computer with only the CPU registers and a hard disk as storage devices. The CPU is directly connected to the hard disk and every piece of instructions and data must be directly accessed from the hard disk to the CPU.[2] Hard disk access time is in the tens of millisecond range while CPU can execute about 4 to 8 million instructions per second. That is about 8,000 instructions per millisecond. This means that CPU must wait some tens of millisecond to run one instruction in this imaginary computer. This means the CPU can only execute about 30 instructions per second because it is slowed down by the hard disk speed. To narrow the speed gap between hard disk and the CPU, we introduce the main memory device in between. However, the main memory access time is still too slow to keep up with the CPU speed. This is why we insert another smaller and faster memory devices L1 and L2 caches in between the memory and the CPU. Recall that as we move toward the CPU, the memory devices sizes gets smaller. That means the amount of instructions and data stored in the hard disk cannot all be loaded in the main memory at the same time. That is true between the main memory and the L2 cache, true for the L2 and L1 caches, true for

[2]Although hard disks are block access device, we assume we engineered that a hard disk that can read/write bytes per access.

the L1 cache and the general purpose registers in the CPU. How are these in-between devices useful? They are useful because the *principle of locality* is in force for every programs executed on computers! There are two types of the principle of locality: the *temporal locality* and the *spatial locality.*

The temporal locality says that the program instruction that was executed recently by the CPU would most likely be executed again. The spatial locality says that the program instructions that are close to the recently executed instructions in source code would mostly likely be executed soon. As an example, consider the C program (let's call it MATRIX) in Figure 3.5. Notice that there are three subroutines called by the main program: `initialize(), sum_matrix()`, and `print_matrix()`. There are three two-dimensional integer arrays of size 1000 by 1000; that is 1,000,000 integers per array. Each array therefore is 4,000,000 bytes, which is approximately 4MB. Because these are static arrays, the compiler would most likely allocate them in consecutive memory locations in the program's (process's) address space. Recall that there are at least four different segments in a process's address space: the text segment, static data segment, dynamic data segment, and stack segment. The text segment is the segment that contains the binary executable code for the program. The static data segment contains the static data structures that were created during the compilation time of the program. The dynamic data segment and the stack segment grows and shrinks as the program executes. For example, the dynamic data segment grows as the program executes the command like `malloc()`. It shrinks when the data structures created by the `malloc()` function are deleted. The example program, MATRIX does not have any `malloc()` command therefore the dynamic segment will stay at size zero. The stack segment is needed for pushing and popping the parameters and the return address of caller when a subroutine is called.

```
#include <stdio.h>    #define SIZE 1000
VOID INITIALIZE(INT MAT1[][SIZE], INT MAT2[][SIZE],
                                        INT MAT3[][SIZE]) {
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
        mat1[i][j] = 1; mat2[i][j] = 2; mat3[i][j] = 0;
   }}} // ---------------------- end of initialize();
VOID PRINT_MATRIX(INT MAT1[][SIZE], INT MAT2[][SIZE],
                                         int mat3[][SIZE]) {
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
      printf("%d ", mat1[i][j]); }
      printf("\n");}
for (int i = 0; i < SIZE; i++)  {
    for (int j = 0; j < SIZE; j++) {
      printf("%d ", mat2[i][j]); }
      printf("\n");}
for (int i = 0; i < SIZE; i++)  {
    for (int j = 0; j < SIZE; j++) {
      printf("%d ", mat3[i][j]); }
      printf("\n");
 }} // ---------------------- end of print_matrix();
VOID SUM_MATRIX(INT MAT1[][SIZE], INT MAT2[][SIZE],
                                        INT MAT3[][SIZE]) {
for (int i = 0; i < SIZE; i++) {
for (int j = 0; j < SIZE; j++) {
   mat3[i][j] = mat1[i][j] + mat2[i][j];
   }}
} // ---------------------- end of sum_matrix();
INT MAIN() {
 int mat1[SIZE][SIZE], mat2[SIZE][SIZE],
                                        result[SIZE][SIZE];
 initialize(mat1,mat2,result);
 sum_matrix(mat1, mat2, result);
 print_matrix(mat1,mat2,result);
 return 0;}
```

Figure 3.5: Program Locality of References

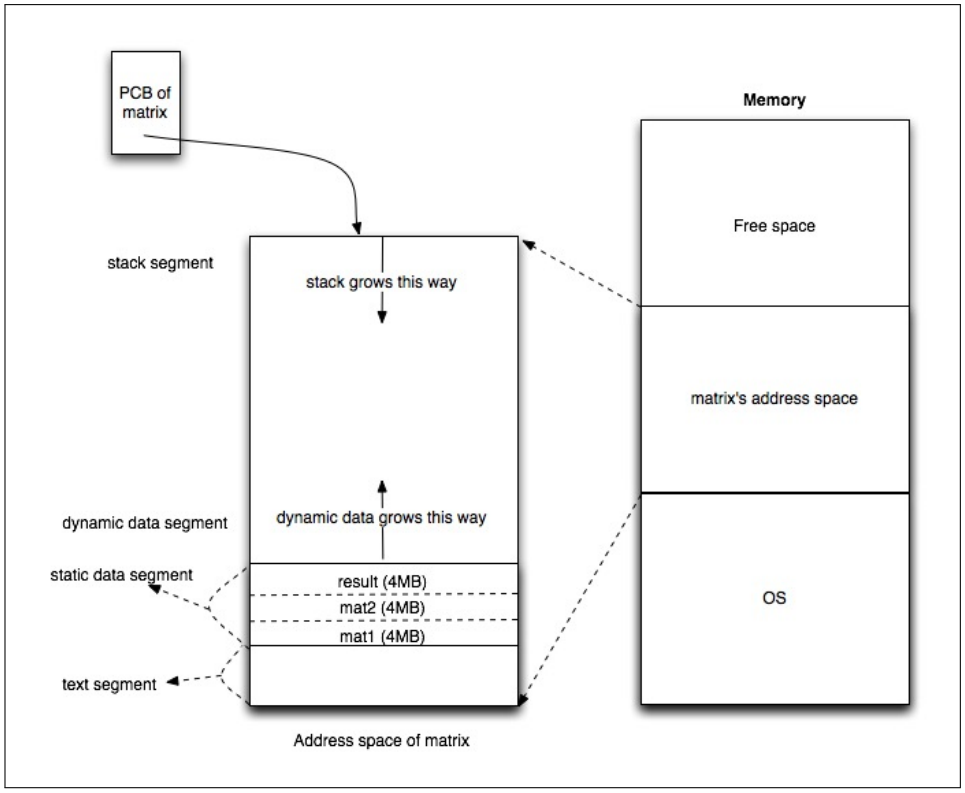


Figure 3.6: An illustration of address space when using a contiguous memory management. The address space here shows four segments, the text segment, static data segment, dynamic data segment, and stack segment.

Now when this program is executed, let's say for now that the L1 cache size is 4MB, the L2 cache size is 8MB, and the main memory size is 16MB. Notice the grayed highlights of program MATRIX. The first highlighted area in INITIALIZE() shows a nested for-loop of 1 million iterations. Within it, three two-dimensional arrays are initialized, which invokes total of 3 million assignment statements. If all three matrices are stored in L1, the fastest memory device next to the CPU registers, the operations would be only bounded by the access speed of L1. Unfortunately, L1 is only 4MB, which is only enough to store one of the matrices. Therefore, if we store `mat1[][]` in L1 cache, when `mat2[i][j] = 2;` is executed, we would have to access L2, which is slower than L1. Furthermore, when `mat3[i][j] = 0;` is executed, we would have to access the main memory, which is slower than L2, because L2 can only contain mat1 and mat2, not not mat3.

```
void initialize(int mat1[][SIZE], int mat2[][SIZE],
                                          int mat3[][SIZE]) {
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
        mat1[i][j] = 1;
    }}
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
        mat2[i][j] = 2;
    }}
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
        mat3[i][j] = 0;
    }}
} // end of initialize()
```

Figure 3.7: Program Locality of References: more efficient code for 4MB L1 cache size

Figure 3.7 shows more efficient code for INITIALIZE(). In this version of INITIALIZE(), we finish the assignment statement of `mat1[i][j] = 1;` a million times first, then perform `mat2[i][j] = 2;`, and then `mat3[i][j] = 0;`, a million times each. Therefore, for each nested for-loop, we access only L1. Do application programmers worry about this? Not necessarily. Fortunately, compilers are well-designed so that they rearrange an input source code to an equivalent and efficient code.

In this code, we can notice that the code and data that was used in recent operations are used again and again (see the nested loop). This is the temporal locality. We can also noticed that mat1 data are accessed recently together because they are spatially (see the address space) closer than mat2 and mat3 data. The same is true for the mat2 and mat3 cases. This is the spatial locality.

Now that we understand the concepts of the locality of references and the memory hierarchy, we need to be able to utilize these concepts in our memory management mechanism as well as in designing an OS.

3.2 Taxonomy of Memory Management Schemes

The memory-management scheme used prescribes how OS will load programs into the main memory and consequently how the program addresses are translated during the runtime of processes running concurrently.

Recall that in Section 1.4.4 and Section 3.1, we briefly discussed the concept of address space. From the application programmer's perspective, the address space of a program (or a process) is always contiguous and the address space starts from address 0. This is the virtual address space as discussed in Section 1.4.5. When compilers generate executable codes, it assumes that every program starts from address 0 and the program is loaded into the main memory as a whole in a consecutive area. A memory management scheme that honors this compiler's view of the programs *loads each program in a consecutive area and for its entirety.*

However, if we do that, we may not be able to run as many program as possible. Why not? Consider a web browser such as Firefox. Users can use the browser to visit web pages with text information or visit web pages to watch movies. When watching movies, the correct codec subroutine must be loaded in the main memory so that the streaming movies can be decoded. However, if the user is only reading some text-based blog, we do not need to load the codec. In other words, when we run a browser program, we do not need the entire browser program every time. This is true for most of the programs you can imagine. In other words, the principle of locality of references is in force here! If we only load the part that we need at any given time, more memory would be available for more programs to be loaded and to run. This means a higher parallelism that fosters higher utilization of computing resources. So naturally, another memory management scheme is to *load the part of the program that is needed at the given time, dynamically.*

Let's consider the main memory snapshot depicted in Figure 3.8. At $t = 1$, there is one contiguous free space of size 3MB. At $t = 2$, `proc2` has finished and relinquished its memory area leaving another 2MB of contiguous free space. However, since the OS is only capable of loading programs in a consecutive area and in their entirety, we cannot load another process, say for example, with size 4MB, even if the size of the combined free spaces will be 5MB, which is more than enough to load and run a 4MB size program. We can solve this problem in two different ways. One way is using so called *memory compaction*. Memory compaction will move the processes loaded in the memory to remove all but one free contiguous space. In this example, we have a 5MB of contiguous space after the compaction. The compaction process however, can be expensive. In a straightforward implementation,

a compaction process will suspend all user program running, and OS will start running to perform a compaction. Imagine using a computer occasionally displaying a message on the screen that says: "Memory compaction in progress. Please wait. This could take a few seconds to minutes!!"

A better way is to *develop a non-contiguous memory management scheme combined with the idea of loading only part of the program.* We will discuss this idea in *paging* and *virtual memory* in Section 3.3.

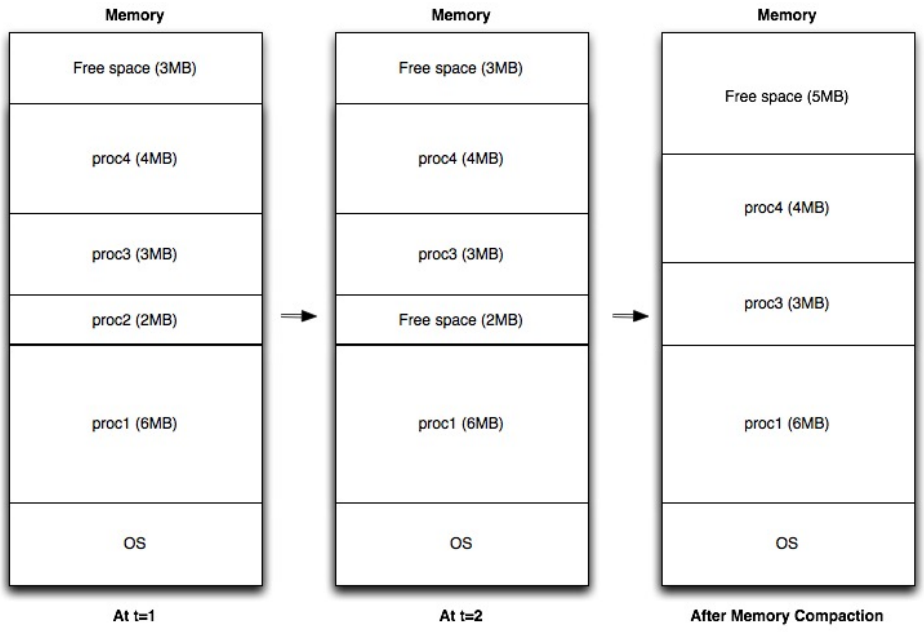


Figure 3.8: Memory external fragmentations when using a contiguous memory allocation scheme

In summary, there are four different ways of memory management: (1) Loading each program in a contiguous physical memory area and loading the entire program before running; (2) Loading each program in a contiguous physical memory area and loading only a small part of the program as needed; (3) Allow loading each program in non-contiguous areas and loading the entire program before running; and (4) Allow loading each program in non-contiguous areas and loading parts of the program dynamically on-demand basis. Case (1) is most straightforward and simple. In fact, case (3) is not common and we will not discuss it. *Overlays* method is a good example for case (3), which is used commonly in game consoles. Case (4) is

paging and virtual memory.

3.2.1 Memory Management Unit and Address Translations

Before we discuss each different memory management scheme, it is important to understand the address translation schemes required for each scheme. Recall that address translation means a virtual to physical address translation because compilers always assume that programs are loaded from address 0 as does the CPU. However, that assumption is not true even in a simplistic case, such as in mono-programming systems, in which only up to two programs are loaded in the main memory at any given time: The OS and the user application program.

```
int main() {
  int i = 0;
  int sum = 0;
  for (;;) {
  for (i = 0; i < 5; i++)
      sum = sum + 1;
  }
}
```

```
main:
    LOAD $i, 0
    LOAD $sum, 0
L1: ADDI $sum, $sum, 1
    INC  $i
    BNE  $i, #5, L1
    JUMP main
```

Figure 3.9: C Program and an Equivalent Assembly Code

Consider the program in Figure 3.9. Notice the assembly program on the right that is equivalent to the C program on the left.

In Figure 3.10, the user application program in Figure 1.13 is loaded in a contiguous area in the physical memory starting from `0x80-1`. The user application program could not be loaded from address 0 because the OS was loaded already from 0 to `0x80`.[3] Because the compiler always assumes that any program will start from address 0 (i.e., a virtual address), the system needs to translate all address references within the program to corresponding physical addresses. Review Section 1.4.5, if needed.

Notice that the program has two branch statements needing address references: `BNE $i, #5, L1` and `JUMP main`. When the compiler produces the binary executable (i.e., the assembly program), it assumed that all the address references happening in the program are within the virtual address

[3] Yes. This is a really small OS for the sake of explanation.

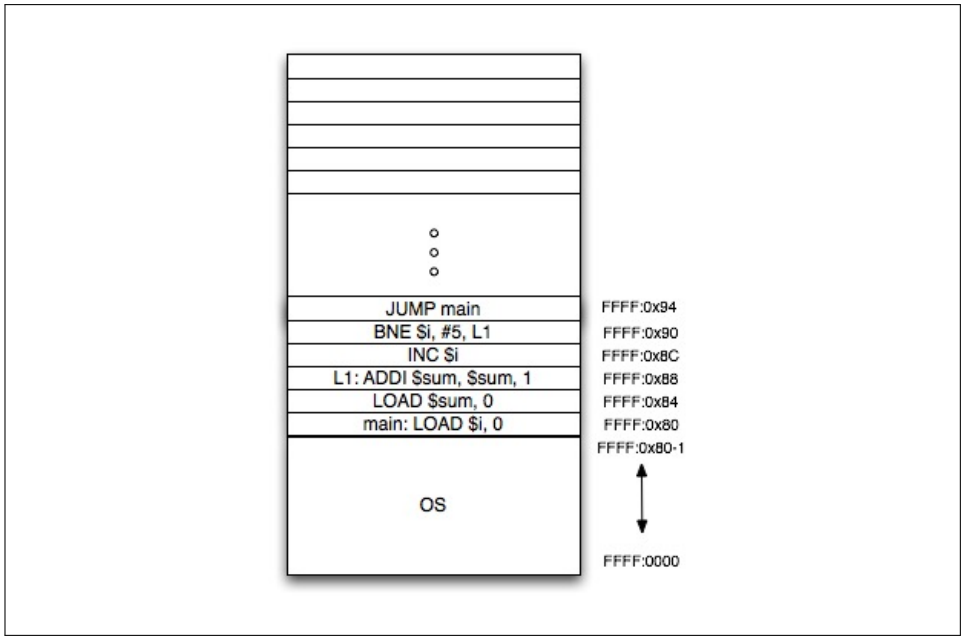


Figure 3.10: A simple memory management with the OS and one user application program loaded.

space, meaning that the starting of the program is address 0, and the program is loaded contiguously in the physical memory. The second assumption is true for Figure 3.10 because the OS used a contiguous memory management scheme. However, the first assumption is not true since the OS couldn't load the program from address 0 because the OS itself was loaded from that address and on. The user program was loaded from the address `0x80`. Each instruction is 4 bytes long, so we have an increment of 4 for each instruction in the figure.

For any given program, we have a large number of condition and branch statements. The CPU also assumes that any program it is currently running starts from address 0 and the address space is contiguous. Figure 3.11 shows this idea. The CPU makes memory access requests with virtual addresses. Them the memory management unit (MMU) translates each virtual address to the corresponding physical address. The MMU is initialized by the OS so that it can properly perform address translations. However, once it is initialized, its work is done with the hardware logic. Since MMU has to work closely with CPU all the time, the MMU is commonly packaged in the

same chip package with the CPU. In a simplistic term, the MMU takes a virtual address from the CPU, translate the address to the corresponding physical address and send the address to the memory through BUS.

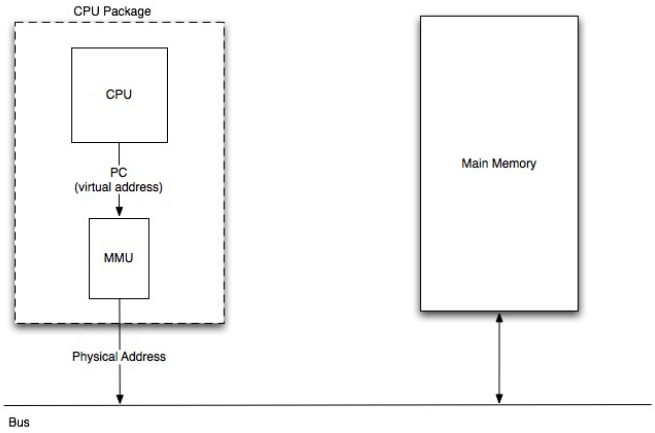


Figure 3.11: A simple MMU for Contiguous Memory Management.

Base and Limit Registers in Contiguous Memory Management

Let's explain how the address translation is being done in the above example. In the MMU, we now have a set of hardware registers called the *base* and *limit* registers. These registers contains the starting address of the currently running process (or thread) and the size of the process. Each program is loaded in a contiguous memory area, so to get the physical address for a given virtual address, we simply need to add the virtual address to the base register's value. The limit register's value plus the base register's value must be larger than the physical address computed. Otherwise the process is attempting to access the address that is outside of its address space.

Figure 3.12 illustrates an example of virtual to physical address translation using a pair of base-limit registers. When a process is selected to run next, during the context switching to the new process, the base and limit values of the process are copied to the base and limit registers in MMU. Currently, the user program that is loaded from `0x80` is running in the example. The PCBs of processes now have two additional pieces of information: the values of base and limit of the processes. Of course, the OS assumes that the processes are loaded in the main memory in a consecutive manner for the base-limit register address translation scheme.

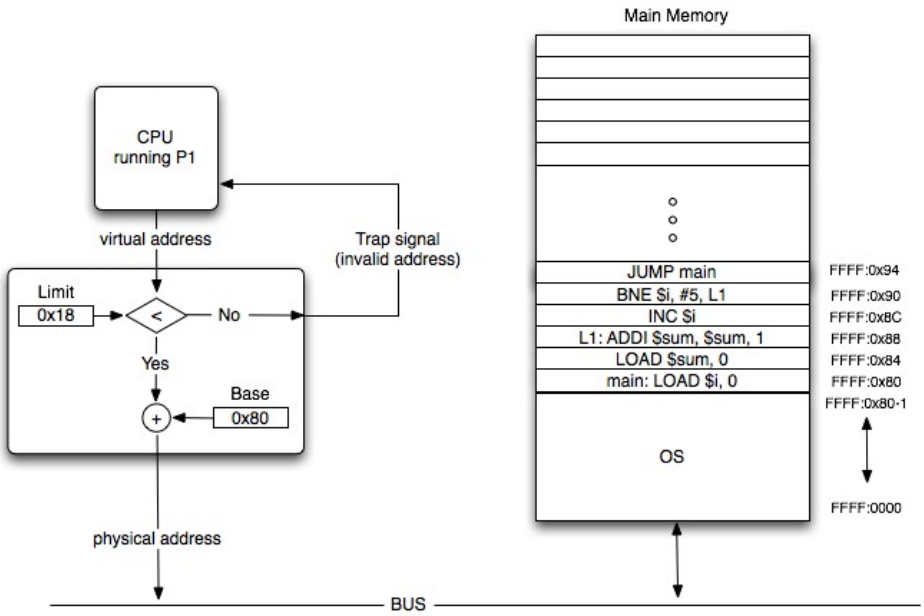


Figure 3.12: A virtual to physical address translation using base-limit register pair. Each process' PCB contains the process' base and limit values. When a process is selected to run next, the base and limit values of the process are copied to the base and limit registers in MMU. Currently, the user program that is loaded from `0x80` is running in the example.

More complicated memory management schemes employing non-contiguous memory allocation methods use a bit more involved logic for virtual to physical address translation. Therefore, the MMU logic for these schemes would be a bit more complicated. We can also use multiple pairs of base-limit registers for each pair that is used to manage a different segment of a process.

3.2.2 Overlays for Game Consoles

Many readers of this book have played video games on game consoles. Nintendo, Playstation, and Xbox are such game consoles. As you play any computer games with multiple stages, you notice that after each stage challenge is successfully played, the DVD or CD drive will run and display the screen message such as "next stage being loaded, please wait." This is the *overlays*. The overlay method allows a program that is larger than the amount of memory available to run. To make overlay work, the application

programer (in our example, the game programmer) must divide the program into multiple parts and program it in such a way that each part is loaded in an order. Notice that the application programmer (the game programmer) must know how much memory is available and appropriately divide the program into parts so that each part alone can fit into the memory available. A natural way of dividing a game would be per stage of the game. Although the previous method of using base-limit registers does not allow us to run a program that is larger than the available memory size, the address translation and memory management were completely hidden from the application programmers. In overlays, that is not true.

How much the application programers need to worry about memory management?

In more advanced memory management technique such as virtual memory with paging and segmentation, which we will discuss later, memory management and address translations are completely hidden from application programmers. All these are done by the OS and hardware. In some simpler systems or older systems, application programmers or even users of the computer systems are forced to get involved in memory management. In an old personal computer operating system, the user must click on the property option for a binary executable program and assign minimum and maximum memory to be allocated when the program is loaded and runing in the system. Virtual memory systems eliminated such worries from users.

3.3 Non-Contiguous Memory Management: Paging and Virtual Memory Systems

The contiguous memory management scheme using base and limit registers can leave small free fragmentations that need to be addressed by memory compaction. Memory compaction is an expensive process. It also assumes the programs are loaded in their entirety, which is a waste of memory space. The scheme also cannot load programs larger than the memory available.

Overlays method was developed so that it can support running larger programs. However, it requires the application programmers to do the memory management of dividing the program in to smaller pieces so that each piece can be loaded at a time. We want to design a better memory management scheme that can load programs in non-contiguous areas after dividing the program into smaller chunks. The scheme should also allow only some

parts of programs to be loaded rather than their entirety. The method also should allow programs that are larger than the physical memory size. We will discuss such a method below.

3.4 Paging, Dividing a Program's Address Space in Equal-Sized Pages

First, let's discuss how we can divide a program into different parts. We already know a program's address space can consist of several segments, for example, text, static data, dynamic data, and stack segments. The sizes of these segments are different from program to program. There is another way of dividing a program's address space, which divides the address space in equal sized *pages*. When we divide a program into pages, unlike dividing it into segments, the boundary between two pages do not have any logical separations as two different segments. For example, if a page size is 4KB and the program's address space size is 33KB, the total number of pages that represent this program is $\lceil 33/4 \rceil = 9$ pages. For each 4KB boundary, the program is divided into a page.

Not only do we divide programs address spaces into pages, we divide the main memory into page frames. Page frames have the exact same size as pages. That's why we call them page frames. When we say "divide the address space" or "divide the main memory," we do not divide them physically. The OS simply draw imaginary boundaries at each page size amount (in our example, for every 4KB). With this scheme, we can only load some subset of pages for each process to memory, even non-contiguously.

Figure 3.13 shows an example of how the main memory is divided into page frames and each process' address space is divided into pages. We assume the page size is 4KB and the main memory is 64KB. This means that there are 16 page frames in the main memory. OS keeps a page table for each process. The page table for a process is a mapping table that keeps information about which page of the process is loaded in which page frame in the main memory. For example, in the figure, page 0 of P1 is loaded in the page frame 6 in the main memory. Also, page 1 of P2 is loaded in page frame 4. 'DISK' means that the virtual page is not loaded in the main memory but can be found in the hard disk partition for virtual memory space. We will discuss this later in detail. 'N/A' means that the corresponding page does not exist. Depending on the process execution behavior, these pages can be created. Think dynamic memory allocation and stacks.

Now let's discuss the entire story about the paging system. Consider

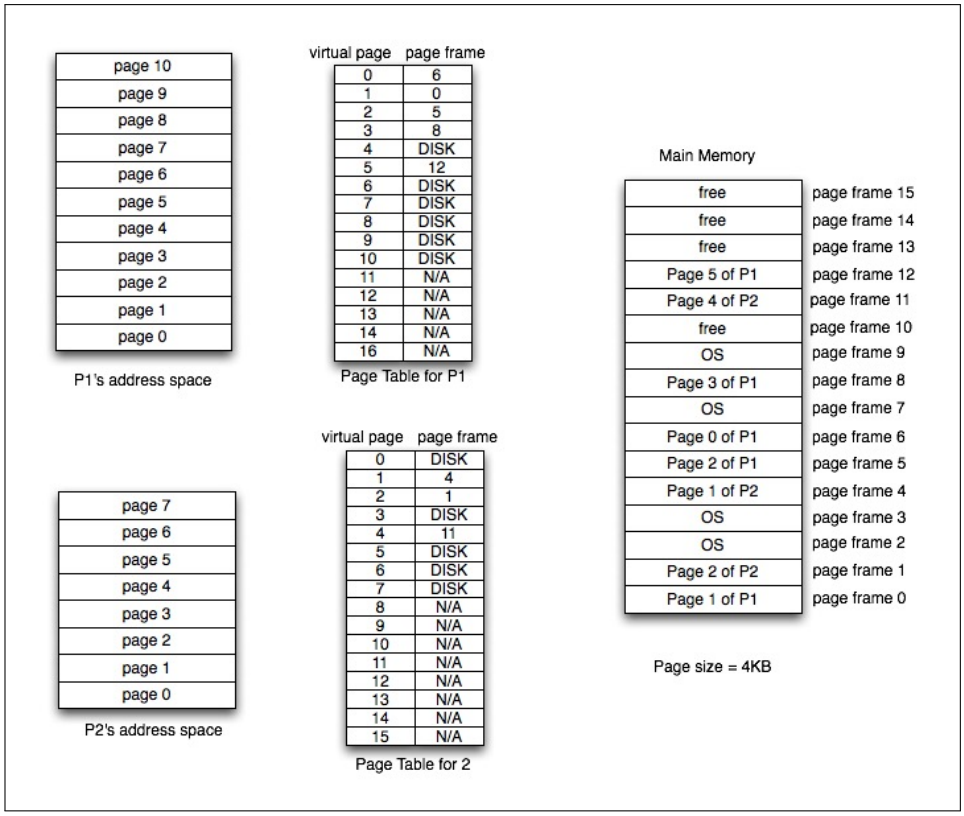


Figure 3.13: An illustration of pages mapped onto page frames. There are two processes: P1 and P2. Each process' page table maps the pages from the corresponding process to the page frame that contains them.

a computer architecture with 16-bit addressing capability. Total memory to be addressed is 2^{16} byes, i.e., 64 Kbytes. The page size is 4KB, defined by the OS. Assume the machine is byte-addressable, which means that the smallest unit of access is a single byte. Given a virtual address, the virtual to physical address translation mechanism first identifies which page frame in main memory contains the page with the specific byte. Once the page frame is found by using the page table of the current process, that specific byte within the page is found using the offset value. The offset value is the amount of bytes from the beginning of the page.

Figure 3.14 shows the overall process of virtual to physical address translations in the computer architecture we defined earlier. In the figure, the virtual address generated by the CPU is `0x1F`, which is in binary `0000 0000`

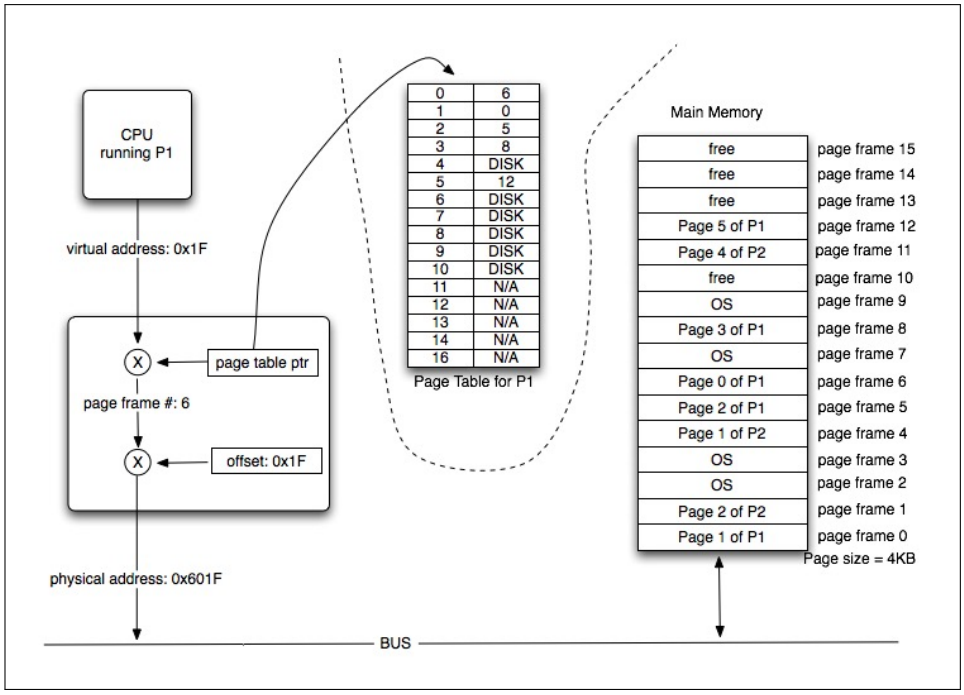


Figure 3.14: The overall process of translation of a virtual address to the corresponding physical address in paging. In this example, the virtual address is `0x1F` and the physical address is `0x61F`. This is because that virtual page number of the address is 0 and the corresponding physical page number is 6. See Figure 3.15 for details.

`0001 1111`. The current process running is P1. Therefore, we need to look at the P1's page table. Since the page size is 4KB, if we divide `0x1F` by `0x1000` $= 0.7_{10}$, i.e., 0.7 in decimal, and take the floor function of the result, which gives 0. That means that page 0 contains this particular byte. Another way to compute the page number is to check the 4 most significant bits for `0x1F`. In other words, `0x1F` in binary is `0000 0000 0001 1111`. Since the page size is 4KB, it requires 12 bits to address bytes within a given page. That is the *offset* within a page.

Figure 3.15 show more detailed explanations on what goes on inside of MMU. Notice that the page table is not inside of MMU. MMU just has the pointer (memory address) of the page table so it can refer to the table.

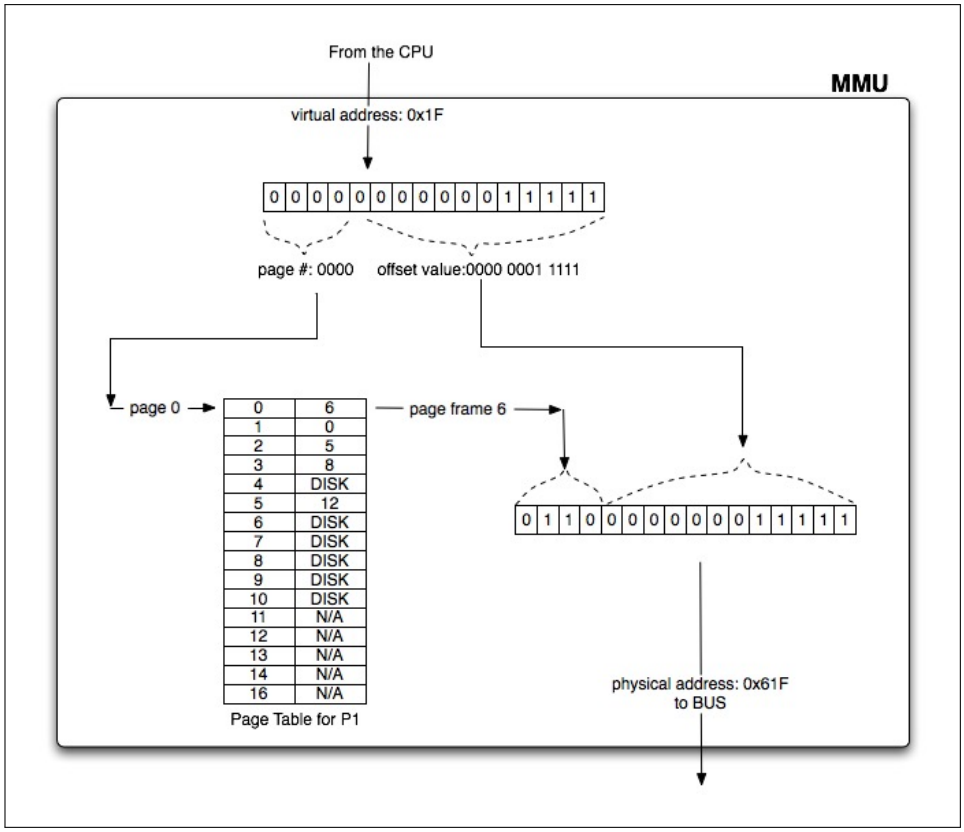


Figure 3.15: More detailed depiction of the operations within MMU for virtual to physical address translation in Figure 3.14. Because the page size is 4KB in this example, the 12 least significant bits are used to address bytes within a page. The remaining 4 significant bits are used to refer to the page number. Using the page number and by referring to the page table, the MMU finds the corresponding page frame for the page. Then the page frame number and the offsets are concatenated to produce the physical address.

If the page to be accessed is not in the main memory, the translation fails. That means the page is not in the main memory but in the hard disk. This is called *page fault*. The *faulted page* must be brought in from the hard disk. When we bring the faulted page from the hard disk, we can bring only the faulted page or we can bring multiple pages along with the faulted page. These other pages being brought in are "related" pages to the faulted page in terms of the locality of reference principle. This way, instead of generating an isolated disk I/O for each page fault, we can utilize hard disk

bandwidth as well as reduce the probability of future page faults by serving one page fault.

3.4.1 Virtual Memory Systems

A virtual memory system allows programs that are much bigger than the main memory size to run. In other words, some portion of the hard disk is designated to act like an extension of main memory. Recall Figure 3.4. In the figure, the CPU is directly connected to the hard disk. Having the main memory in between the CPU and the hard disk is much like having a cache memory between the CPU and the main memory.

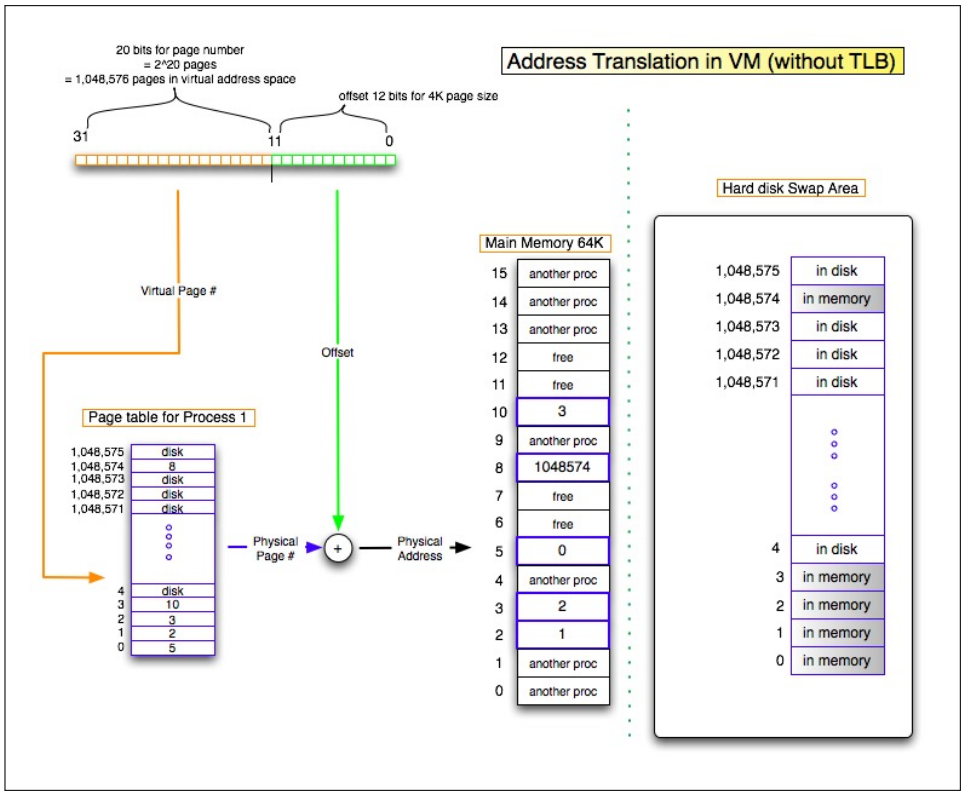


Figure 3.16: An illustration of a virtual memory system using paging. The example shows for a 32-bit CPU architecture.

Figure 3.16 shows a virtual memory system for a 32-bit CPU architecture using paging. Pages for active processes are stored in the hard disk's swap area (i.e., virtual memory area). As pages are needed for computations, only

needed pages are brought into the main memory. The swap area is much larger than the main memory so it can accommodate very large processes. The figure also shows an additional example of virtual to physical address translation for a 32-bit CPU architecture. Since the size of page is 4KB, 12 bits are required for offset. We then have 20 bits for addressing page numbers, which is $2^{20} = 1,048,576$ pages.

3.4.2 Performance Issues in Paging

There are performance issues we need to address in paging: the speed and the space (memory space) issues. Memory accesses are slower, because the added steps in virtual to physical address translation due to the access to the page table. Accessing a page table is additional memory access. Therefore, paging could increase the memory access time to at least twice as long. This is a delay in address translation. What makes it worse is the case when the desired page is not found in main memory so that the page must be brought in from the hard disk. In this case, the access time can be increased by the slowest device within the path, which is the hard disk. This is a delay in data access.

In terms of memory space, notice that a page table is needed for each process. In our example, there are total 16 entries in the page table. If each entry is 8 bytes, each page table is $8 \times 16 = 128$ bytes. If we run about 50 processes at the same time, $50 \times 128 = 6,400$bytes = 6.4KB. One may think this is not much overhead. But modern CPU's address capability is usually 64 bits. Assuming the same page size, 4KB, we have 64-12 = 52 bits for referring to page number, which is 2^{52} entries in a page table. For 8 byte big page table entries, one page table size is $2^{52} \times 8bytes = 2^{55}$ bytes, which is 32,768 terabytes for one page table. There is no computer with the main memory size that big. We must address this issue.

Speeding up Address Translation

In order to speed up the address translation, we can build a cache memory that stores the most likely address translation that would be needed in near future. Because of the locality of reference principle, we can store some `<virtual address, physical address>` pair in the cache and look up the physical address without referring to the page table. This cache is called Translation Lookahead Buffer (TLB). This is a special hardware logic component using an associative memory, so that the access time is fast. The typical size of TLB can be about 1,024 entries. If the address translation

is found in the TLB, we bypass the page table lookup and produce the corresponding physical address.

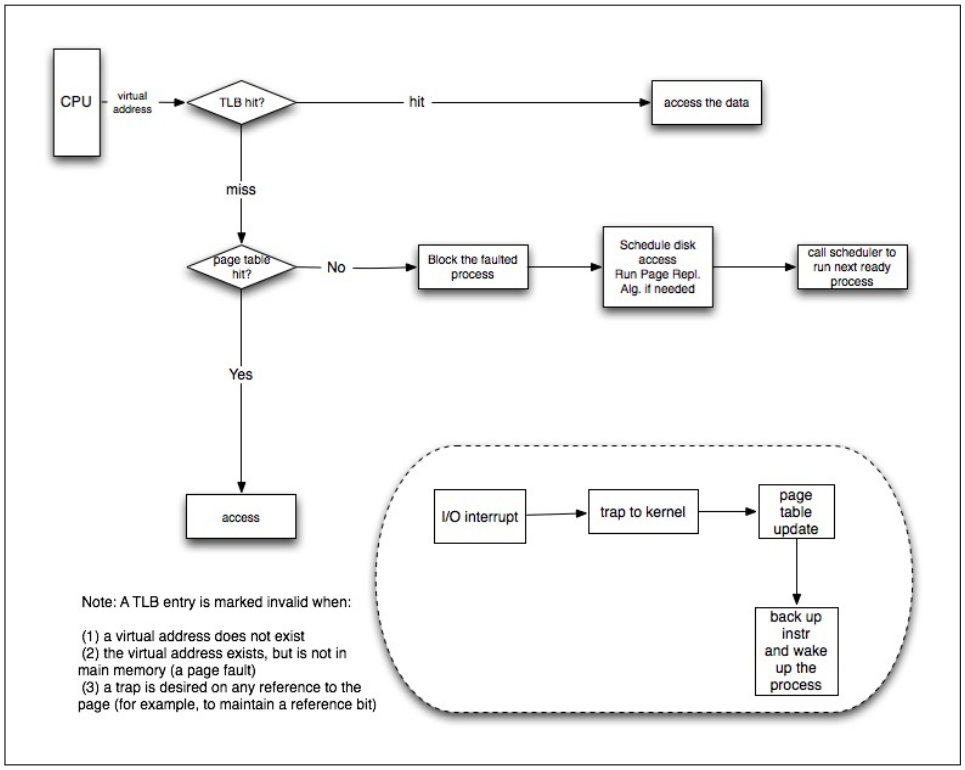


Figure 3.17: Address translations using TLB and page table. If the virtual-physical page pair is found in TLB, we can access the data (having its physical memory address) immediately without additional delay. If the address pair is not found in TLB, the translation must be done using the page table.

Figure 3.17 shows the address translation process using TLB and the page table. MMU will first attempt to translate the virtual address through TLB; if no valid translation entry is found, it will then access the page table to do the translation. If the TLB hit ratio is high enough, meaning that the probability of finding the desired `<virtual address, physical address>` pair is high enough, the address translation in paging systems will be fast enough to be comparable to non-paging systems. Notice that in case of a page fault, an I/O request must be scheduled and the process that caused the page fault must be suspended until the desired page is brought into the

main memory. When the page is brought in, the page table must be updated to reflect that information; in other words, that particular page now is in the memory so the page table must reflect that!

Speeding up Memory Access (i.e., page access)

After getting the right physical address from the translation phase, the OS need to access the data or instruction stored in the physical address. There are two possibilities: (1) the page that contains the desired data/instruction is stored in the main memory; (2) the page that contains the desired data/instruction is not in the main memory and it must be brought in from the hard disk.

In case (1), we do not have an additional overhead and the OS can go ahead and access the data/instruction. Case (2) can be further divided into two different scenarios. First, there is at least one available page frame so that the desired page can be brought in from the hard disk without additional steps. The desired page can be brought into one of the free page frames. The second scenario is that there is no available page frames and we must find a page in one of the frames to be replaced by the desired page. Which page in which page frame should be selected to be replaced? OS uses somthing called a *page replacement algorithm* to choose the "victim" page. The victim page should be the one that would be least likely to be accessed again in near future. Also a victim page is not 'dirty,' which means that it has not been changed since it was brought into the main memory, therefore, there is no need to write it back to the hard disk to be updated; the OS can simply overwrite it with the desired page coming from the hard disk.

Page replacement algorithms are predictive algorithms that attempt to predict the pages that are least likely to be used in the near future. We discuss several page replacement algorithms below. By using a smarter page replacement algorithm, OS can improve the speed of memory access. OS can also improve memory access by caching pages that are most likely to be used again in L1 and L2 data cache,

Reducing the Memory Overhead of Page Tables

As we discussed above, for a 64-bit architecture CPU, the amount of page table memory overhead is extremely expensive. There are two ways of alleviating this problem. The first is using *multi-level paging*, which divides one big page table into several smaller page tables. Store most part of the page table in the hard disk and keep only a small part of the page table in

memory.

For example, for a 64-bit machine with 4KB page size, we have 52 bits for page table entry access. We divide these 52 bits into four 13-bit groups. Each 13-bit group refers to a smaller page table with 8,192 entries. If each entry is 8 bytes, we have 64 KB, which is much smaller than 32,768 terabytes. Because the locality of reference principle is in force here, we only need to bring a small number of these small page tables in the main memory, greatly reducing the memory overhead.

Another way for reducing the memory overhead of page tables is redesigning the paging scheme from scratch. The inverted page table (IPT) scheme would only require one IPT for the entire system rather than one page table per process as in the regular page table scheme.

Instead of searching for the page frame that contains the page desired, the IPT scheme searches through page frames to find the desired page. Therefore, the number of entries in the IPT is the amount of physical memory size divided by the page size. If the memory size is 16 GB, with 4KB page size, there are 4,000,000 entries in the IPT. If each IPT entry is 8 bytes, the size of the IPT is 32 MB, which is only 0.2% of the total memory. Furthermore, there needs to be only one IPT for the entire system. In the IPT scheme, the virtual address format is a bit different from the traditional paging. A virtual address for IPT consists of the `<current process id, page #>` pair. Each IPT entry also consists of the pair `<process id, page #>`. For example, if the entry 0 of IPT contains `<5, 7>` pair, it means that the page frame 0 contains process 5's virtual page 7. If the virtual address matches with one of the entries in the IPT, that is the page desired. Searching the 4,000,000 entries in the above example can be slow if done linearly. We can use a hash function of $h(currentprocessid, page\#)$ that yields the page frame number to speed up the search.

3.5 Segmentation with Paging (For most modern OSs)

Recall the address space discussed earlier. We discussed four segments in the address space: the text segment, the static data segment, the dynamic data segment, and the stack segment. Both the text and the static data segments are fixed in sizes. However, the dynamic data segment and the stack segment can grow and shrink during runtime. During the compilation time it is not possible to predict how big these two segments will grow when the program is executed. This is because we can't always predict the

program behavior statically. The dynamic data segment will grow whenever the running program calls statement like `malloc()` in C or `new Classname` in Java. The stack segment will grow whenever a subroutine call is made to push the arguments of the subroutine and the return address to the program stack.

If all these four segments share one virtual address space, the dynamic data segment and the stack segment can eventually collide. Segmentation with Paging allows each segment to be treated as a separate virtual address space. Therefore, each segment can grow and shrink at will, as far as the virtual memory space allows and the addressing capability of the CPU allows. Then each segment is divided into pages and the address translation scheme will first identify the segment of the virtual address, then the page.

Figure 3.18 shows a comparison between pure paging and segmentation with paging schemes. There is only one virtual address space in pure paging. All four segments are linearly arranged within the only one virtual address space. In the segmentation with paging scheme, each segment has its own virtual address space, allowing them to grow or shrink. To translate a virtual address, first find the right segment using the first two most-significant bits, and then identify the page number. Using the virtual page number and the page table for the current process (the page table is not shown here), we can find the physical page frame that contains the virtual page for segment 10_2. The virtual to physical address translation is very similar to the pure paging scheme.

3.6 Page Replacement Algorithms

We briefly discussed a scenario in which which a page fault occurs, and we need to bring the desired page from the hard disk to the main memory. If there is no unused page frames available in the main memory, we need to find a victim page to be replaced. Which page in the main memory should be chosen to be the victim page? The answer is the page that would be least likely to be used in the near future among all the pages in the main memory. Although this sounds wonderful, it involves predicting future, which we cannot do accurately in a consistent manner. However, we could observe some past tendencies of each page's "usefulness" and make an educated guess. Each page replacement algorithm except the Random algorithm, attempts to utilize some information about the pages and make such predictions. Most of these algorithms have common assumptions: the pages that were used frequently or many times, or both have higher probability for getting

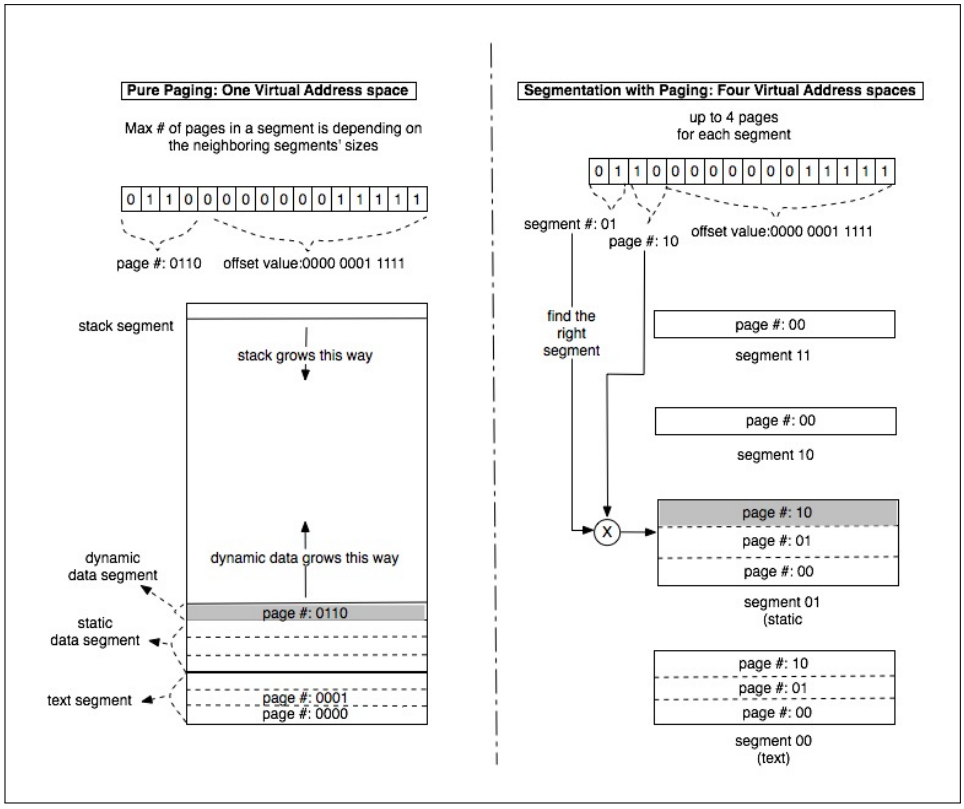


Figure 3.18: A comparison between pure paging and segmentation with paging schemes. There is only one virtual address space in pure paging. All four segments are linearly arranged within the only one virtual address space. In the segmentation with paging scheme, each segment has its own virtual address space, allowing them to grow or shrink.

needed again and soon.

Although page replacement algorithms are developed originally for paging systems in OS, these algorithms also inspired algorithms in other areas including user interface and robotics. In user interface applications, for a mobile phone, most frequently called numbers are displayed first. This is similar to the assumptions made by page replacement algorithms.

All page replacement algorithms starts with the question: How to choose the "best" one to replace? In order to answer the question, all page replacement algorithms keep track of some information about each page. When the algorithms need to choose a victim page, they use the collected information

to do the selection. The followings are some common reasonings done by most of page replacement algorithms:

- What are the things done at the time of a memory reference for pages (read/write)?
  - What kind of information is kept?
  - Is the information kept forever and accumulated or is it cleared occasionally?
  - Is there anything done when a clock interrupt happens?
- What are the operations done at the time of a page fault?
- How do we use and manipulate the information?

Anther analogy to page replacement algorithms is music CD collection storage methods. If you have, say 1,000 music CDs but your CD rack can contain only 100 CDs, which 100 out of the 1,000 CDs would you store in the CD rack? One method is to choose 100 most frequently listened CDs and store them in the CD rack and store the rest of them in the basement of your house. But what if you want to listen to one or more CDs stored in the basement today? You would need to bring some CDs in the CD rack to the basement and bring the CDs you want to listen to the rack. Which CDs in the CD rack should you bring to the basement? In each page replacement algorithm we discuss below, imagine each page is a CD in this analogy, page frames are the CD rack (with 100 capacity), and hard disk space is the basement space. Also, think about how to keep track of which CDs are listened to "frequently"?

3.6.1 Random

A random page replacement algorithm chooses a random page frame and replaces the page stored in the chosen page frame. When the number of page frame is relatively small, a random algorithm may work comparably well as a more complicated algorithm. For example, if we need to perform a replacement algorithm for L1 or L2 cache memory, the number of pages that are stored in these devices is small. The advantage of the algorithm is its speed and the simplicity. Because it does not need to keep any past information about page usages, no additional memory overhead exists either.

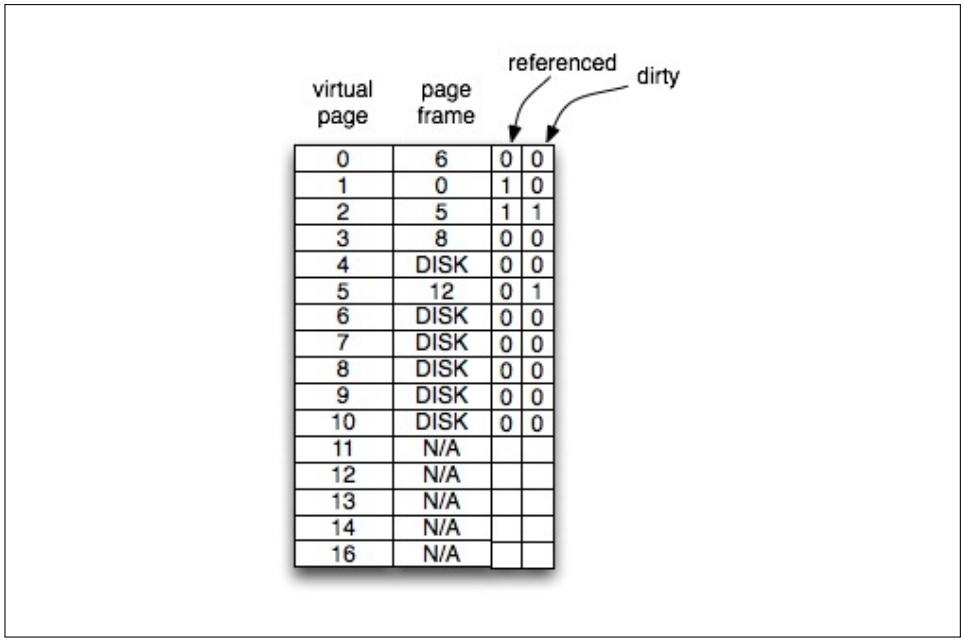
3.6.2 First-in First-Out

The FIFO (First-in First-Out) algorithm keeps track of the order of the pages as they are brought into the memory. Then it replaces the oldest one

in the memory. As an implementation, we can use a sorted linked-list to keep track of the order. This algorithm may work well in some cases, but when older pages are more frequently used, this algorithm will replace these frequently used pages only to bring them back to memory in the near future. In this scenario, page fault can occur frequently. In general, when page faults occur too frequently causing OS to take care of the faults too often, we call this phenomena *thrashing*. When trashing happens, user programs get almost no CPU time because OS runs most of the time to serve page faults. Thrashing can happen in many different situations, so this phenomena is not only related to the FIFO algorithm.

3.6.3 Not Recently Used (NRU)

In NRU, two additional bits are used to keep track of each page's usage pattern. These two bits can be part of the page table entry for each page. Figure 3.19 shows the usual virtual and physical page number pairs; it also shows the two bits: Referenced bit and Dirty bit. If a page was referenced since the last clock interrupt, the bit is set. When a clock interrupt occurs, the Reference bit is reset. A page's dirty bit is set when the page has been modified (by the CPU most likely). If a page is selected to be replaced and its dirty bit is set, we know the page must be written back to the hard disk for updating. If the dirty bit is not set, the page can be simply replaced by the incoming page from hard disk.



virtual page	page frame	referenced	dirty
0	6	0	0
1	0	1	0
2	5	1	1
3	8	0	0
4	DISK	0	0
5	12	0	1
6	DISK	0	0
7	DISK	0	0
8	DISK	0	0
9	DISK	0	0
10	DISK	0	0
11	N/A		
12	N/A		
13	N/A		
14	N/A		
16	N/A		

Figure 3.19: A page table with referenced and dirty bits. For each access to a page, the page's page table entry is updated appropriately including these bits.

Notice that when the dirty bit is set for a page, the reference bit for the page is also set. There can still be the situation in which a page's reference bit is cleared but its dirty bit is set. This is because the reference bits are cleared at every clock interrupt.

So how does the algorithm select a victim page? It sorts pages in terms of their reference and dirty bit first. A page with (r:0, d:0) is first searched, and if such page has been found, that page is the victim page. If there is no such a page, the algorithm will continue to search for a page with(r:0, d:1) to choose. If the search is not successful, a page with (r:1, d:0) is searched. Finally, a page with (r:1, d:1) is chosen if all other searches fail.

Notice that it is debatable to choose between pages with (r:0, d:1) and (r:1, d:0). The second case indicates that the page was more recently used than the (r:1, d:0) case. However, if we choose to replace a page with (r:0, d:1), we need to write the page back to the hard disk to update. This is an additional overhead to consider.

3.6.4 Least Recently Used (LRU)

LRU keeps track of how many times each page is used. A couple of different implementations exists: one using a counter associated with each page table entry and the other using a linked list of page numbers that is updated for each page reference. LRU is clearly expensive because for each memory reference, OS or a designated hardware component must update information about page usages.

When a page fault occurs and a victim page needs to be chosen, LRU will choose the page with the smallest counter value associated, if counters are used for implementation. If a linked list is used, the page that is at the end of the linked list will be replaced.

3.6.5 Not Frequently Used (NFU)

Because LRU is very expensive, there are several algorithms that approximate it. By "approximating," we mean that instead of incrementing the counters for every page reference, an approximation algorithm can increment the counters less often. If a linked list implementation is used, updating the linked list is done less frequently.

For example, NFU uses a reference bit. If a page is referenced since the last clock interrupt, at the current clock interrupt the counter is incremented by 1. If the reference bit is clear, the counter for the page is not incremented. Notice that between two clock interrupts, a page can be referenced multiple times, but the counter will only be incremented by 1, therefore it is an approximation of LRU. The obvious advantage of NFU algorithm is its comparatively lighter overhead.

3.6.6 Two-handed Clock

All of the above algorithms attempt to address a page fault as it happens when a victim page is needed. The two-handed clock algorithm tries to leave a certain number of page frames available at all times so that when a page fault happens, there is no need to call a page replacement algorithm reactively.

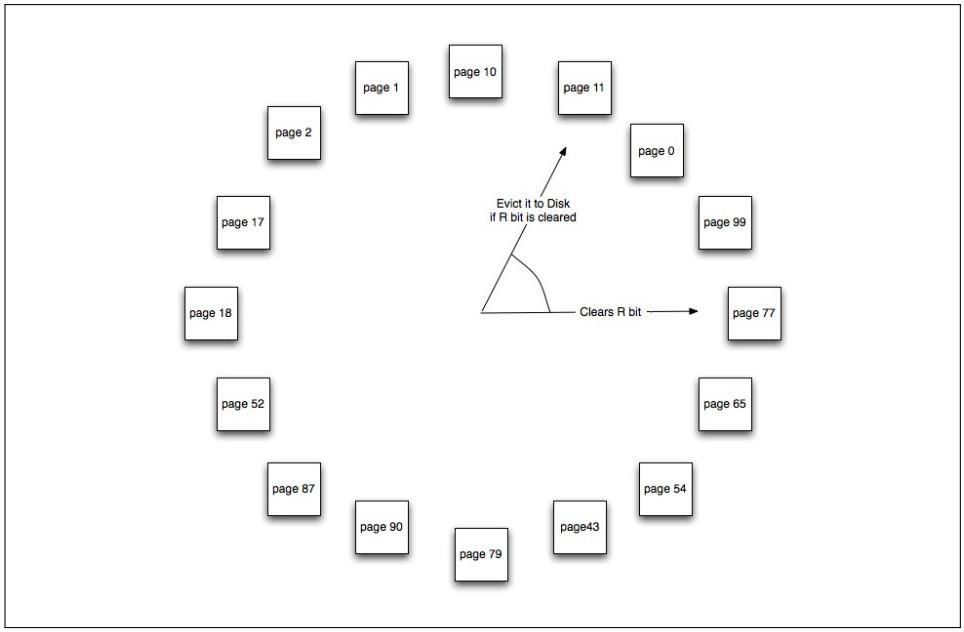


Figure 3.20: An illustration of the two-handed clock algorithm. The algorithm keeps a circular linked list of pages in the main memory. The first hand reset's the R bit of the page it is pointing to and the second hand evicts the page if the page's R bit is cleared. When a page is referenced, the page's R bit is set.

There are two hands turning in a clock-wise fashion as shown in Figure 3.20. The first hand clears the R bit as it points to a page, the second hand will evict the page from the memory to hard disk if the R bit of the page it is pointing to is cleared. At each page reference, the corresponding page's R bit is set. Notice the angle between the hands. If the angle is really wide, the second hand would take longer to point to a page after the first hand. This means there is more time for the pages to get their R bits set. If two-hands are overlapping (the angle is zero), then every page is evicted.

The two-hand clock algorithm can be used by the page daemon process as explained below, because it can be used to ensure a certain number of page frames are always available by adjusting the angle.

There are many more page replacement algorithms. Among them, the Working Set (WS) page replacement algorithm is interesting. The WS algorithm attempts to find out the set of pages that are needed currently and in the near future. The algorithm directly attempts to utilize the locality of references principle.

3.7 Other Concerns in Virtual Memory

There are many other details in virtual memory system that we have not discussed. Discussing all of such topics is beyond the scope of this book. However, we mention a few of the important topics below.

3.7.1 Page daemon and Pre-paging vs. Demand paging

The page daemon is a system process that runs regularly or when there are not enough free page frames. When it runs, it can run the two-handed clock algorithm to ensure that a certain number of page frames are free and available, more or less, all the time. It can also perform *pre-paging* that predictively brings pages that would be most likely used in the near future. This will reduce the page fault frequency, the number of page faults in the system per unit time. When the page fault frequency reaches a certain threshold, the system must suspend some non-critical processes temporarily and allocate more page frames to remaining processes so that they can finish. Once these processes are done, the suspended processes can run again with more memory available to them. When we suspend a process, we store the snapshot of the process state in the hard disk. This is called *swapping out processes.*

The opposite policy to pre-paging is demand-paging. In demand-paging, the policy only brings pages when they are needed specifically by corresponding page faults.

3.7.2 Other Issues

Disk Map: Finding the Page in the Hard Disk

When a page fault occurs, the desired page must be brought in from hard disk to the main memory. We need a method to find the desired page in the hard disk. A disk map is used to keep track of where to find a specific page in the hard disk's virtual memory area.

Instruction Back Up, Involuntary I/Os

When a page fault occurs, the process cannot continue to run until the desired page is brought in from the hard disk. Notice that in this case, a page fault caused a hard disk I/O. The process caused a page fault and consequently it needs to be blocked and wait for the I/O operation to be completed (which means the desired page is now in the memory). After the

I/O is completed, an interrupt is raised to notify the CPU that the I/O for the page fault has been completed. The CPU then runs the page fault I/O exception handler, in which the current processes's program counter is one step backed up to attempt to rerun the instruction that caused the page fault. The process is now waken up and inserted back in the ready queue. When the scheduler picks the process, the process will start running from the instruction that caused the page fault.

Because a page fault causes an I/O but the programmer of the process didn't intend the I/O, we call this an *involuntary I/O*. A page fault causes an involuntary I/O to the process that caused it!

Exercises

Ex. 3.1 — We discussed a contiguous memory management scheme using a pair of registers: the base and limit registers. What happens if we only use the base register but not the limit register?

Ex. 3.2 — One part of memory management is to determine where in the main memory each program to be loaded. This is decided by the operating system. During the compilation time, the location for a program to be loaded is not known. What is the assumption of the compiler about where the program is to be loaded?

Ex. 3.3 — Precisely distinguish between the virtual memory space of a process and the physical memory of a process.

Ex. 3.4 — Can MMU be implemented in software? What are the consequences of that. Why?

Ex. 3.5 — The CPU (most of them) assumes that programs are running from address 0 and the address space of the programs are contiguous. MMU enable the CPU make such an assumption. Consider a memory management for a system that only allows one application program to be loaded in the main memory and run. This means that at any given time, only the OS and one application program are loaded in the main memory. If we want to support the CPU's assumption without an MMU, how would you load the program and OS in the main memory? Explain your idea by drawing the memory map.

Ex. 3.6 — Memory hierarchy is given by the speed of memory devices and the size of the devices along the hierarchy. What if we build a giant memory device made of the fastest materials available that is as big as some tera bytes? Would that eliminate the need for having intermediate memory devices as in memory hierarchy? Why or why not?

Ex. 3.7 — Read Edgar Dijkstra's "Go to statement considered harmful," which appeared in *Letters to the Editor*, in the Communications of the ACM 11, 3, March 1968. What are the consequences of this article to the memory management in terms of memory hierarchy?

Ex. 3.8 — Read Edgar Dijkstra's "Go to statement considered harmful," which appeared in *Letters to the Editor*, in the Communications of the ACM 11, 3, March 1968. Explain why writing subroutines or iteration-blocks small

is important in terms of speeding up memory access in terms of memory hierarchy?

Ex. 3.9 — Read Edgar Dijkstra's "Go to statement considered harmful," which appeared in *Letters to the Editor*, in the Communications of the ACM 11, 3, March 1968. How does this article relate to the locality of reference principle?

Ex. 3.10 — Read about "von Neumann bottleneck." How is this idea related to the memory hierarchy?

Ex. 3.11 — Explain von Neumann's "stored-program computer" model. How would this model be inefficient in terms of program execution time?

Ex. 3.12 — Figure 3.7 shows an example of an efficient code that utilizes the locality of reference principle. Come up with another such example code.

Ex. 3.13 — One of the goals of memory management is to run programs that are larger than the actual available physical memory. Overlays is one such method. Programmers for game console machines use this method often. Explain why it is natural to use overlays for game console machine programming.

Ex. 3.14 — Overlays is not transparent to application programmers, which means that application programmers must decide and program in which overlays are to be loaded and when. Discuss how we can design a memory management scheme that can run programs larger than the size of main memory transparently from application programmers.

Ex. 3.15 — Figure 3.12 shows the MMU algorithm for a contiguous memory management scheme with a pair of base and limit registers. Why do we need the comparison of inequality within the MMU?

Ex. 3.16 — In a paging system with page size that is 8KB, if the process size is 145KB, how many pages are needed in its address space?

Ex. 3.17 — Consider a system with a 64-bit CPU. If page size is 8KB, what is the maximum number of pages there can be for a process? What is the maximum number of entries in a page table? What is the number of bits in the address line to refer to the pages. What is the number of bits required to address the offset within a page?

Ex. 3.18 — Consider a computer with a CPU with 64-bit addressing capability, a CPU cache, TLB with 32 entries, Main Memory of 512 MB, and 100G hard disk using virtual memory scheme with 1-level linear paging. Page size is 16 K. Assume TLB hit ratio is 0.3, assuming page table for the process is always loaded in the main memory. If all of the desired data items are in the memory but not in the cache, on average, out of 100 memory references, how many times do we access main memory? Show your work

Ex. 3.19 — Explain how Two Handed Clock Page Replacement Algorithm, Pre-Paging, and Page Daemons work together. Also explain how using these three mechanisms together would help disk bandwidth utilization.

Ex. 3.20 — There is a page replacement algorithm called the *Working Set replacement algorithm*. Read about it and explain how it works.

Ex. 3.21 — Briefly discuss the trade-offs between a smaller page size and a larger page size.

Ex. 3.22 — Why would one use NFU over LRU?

Ex. 3.23 — NFU and LRU both have a problem of not forgetting. In other words, if a page is used very frequently during the past but no longer needed, these methods will still keep the page in the memory. How would you directly improve these algorithms to address the problem?

Ex. 3.24 — Virtual memory management scheme allows large programs that are larger than main memory size utilizing hard disk space. Usually paging is a requirement to a virtual memory scheme. Describe how you would use only segments to implement a virtual memory management scheme.

Ex. 3.25 — For a 8-bit CPU with a 32-bytes page size, if the CPU generates the virtual address `129` in decimal, which is `10000001` in binary, what is the page number and offset of this address? Assume the machine is byte-addressable.

Ex. 3.26 — For a 8-bit CPU with a 32-bytes page size, if the CPU generates the virtual address `129` in decimal, which is `10000001` in binary, what is the page number and offset of this address? Assume the machine is word-addressable and one word is 4 bytes in this machine.

Ex. 3.27 — For a 16-bit CPU with a 1KB page size, if the CPU generates

the virtual address `129` in decimal, which is `10000001` in binary, what is the page number and offset of this address? Assume the machine is byte-addressable.

Ex. 3.28 — In this chapter, I mentioned that page faults are "involuntary I/Os." After an I/O has been completed, an interrupt is raised to wake up the process waiting for the I/O to complete. In paging system, the faulted process must attempt to execute the faulted instruction again. This is done by adjusting the PC value of the process to the faulted instruction. However, for an I/O command (such as printf(), etc), after the I/O request has been completed, the process should not run the I/O command again. Describe the OS component that can distinguish the difference between an I/O caused by a page fault and an I/O caused by an I/O command.

Ex. 3.29 — For a 32-bit CPU with a 1KB page size using the segmentation with paging scheme, if we utilize 8 bits for referring to the segment, how many virtual addresses are there for each process?

Bibliography

[1] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[2] Calin Cascaval, Evelyn Duesterwald, Peter F. Sweeney, and Robert W. Wisniewski. Multiple page size modeling and optimization. *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 0:339–349, 2005.

[3] Robert C. Daley and Jack B. Dennis. Virtual memory, processes, and sharing in multics. *Commun. ACM*, 11(5):306–312, May 1968.

[4] P. J. Denning. Working sets past and present. *IEEE Trans. Softw. Eng.*, 6(1):64–84, January 1980.

[5] Peter J. Denning. Thrashing: Its causes and prevention. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 915–922, New York, NY, USA, 1968. ACM.

[6] Peter J. Denning. Virtual memory. *ACM Comput. Surv.*, 2(3):153–189, September 1970.

[7] Edsger W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Commun. ACM*, 11(3):147–148, March 1968.

[8] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. *SIGOPS Oper. Syst. Rev.*, 29(5):201–212, December 1995.

[9] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[10] Jason Nieh, Christopher Vaill, and Hua Zhong. Virtual-time round-robin: An o(1) proportional share scheduler. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, pages 245–259, Berkeley, CA, USA, 2001. USENIX Association.

[11] Mohan Rajagopalan, Brian T. Lewis, and Todd A. Anderson. Thread scheduling for multi-core platforms. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, HOTOS'07, pages 2:1–2:6, Berkeley, CA, USA, 2007. USENIX Association.

[12] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[13] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109 – 176, 1997.

4

Concurrencies: Synchronization and Mutual Exclusions

When application programmers write multiprocess (or multi-threaded) programs to achieve parallelism, they must make sure the order of executions among the processes or threads happen as intended. Each process or thread is a separate entity and the scheduler will treat them so. Remember that the scheduler is a part of OS and therefore application programmers have no control over which process (or thread) the scheduler will pick to run next. This sounds like application programmers do not have much control over the order of execution.

However, OS provides some programming commands that application programmers can invoke to manually control processes and threads. This control can be done by stopping a process or thread for a certain condition using appropriate system calls; of course programmers can only control the ones that are owned by them. Here is a list of some system calls that can be used to control process execution:

- `waitpid(), sleep()` etc.: blocking a process/thread that calls the system call.
- `yield()`: context-switch out the process/thread and put the process in the ready queue. This causes the scheduler to run another process/thread in the ready queue.

- `exit()`: finish the current process/thread. This causes the scheduler to run another process/thread in the ready queue because the CPU needs something to run.

- *Semaphores, Locks*, and other synchronizations and mutual exclusion primitives. We will explain these below. These are the main topics of this chapter.

- *Monitors*: a programmer friendly way of achieving synchronizations and mutual exclusions.

We have just introduced some new terms. We will explain these in detail in this chapter.

When multiple processes or threads share some variables or data structures, because a certain order of execution is not guaranteed, unless the programmer uses special system calls (semaphores or locks) or some special features available in concurrent programming languages (Java synchronized, concurrent C++, etc), the resulting value of the variable or the data structures are non-deterministic. Non-deterministic means that the results are not always predictable but one of several possible outcomes. These processes/thread sharing data structures or variables must be *mutually exclusive* when accessing these shared resources. That is, only one process/thread can access a shared resource at any give time. This is *mutual exclusion*. This chapter will discuss this topic in depth and provide solutions to this problem.

One last thing before we continue on this chapter. All the concepts discussed in this chapter are relevant to both processes and threads. At times, we will use these two terms interchangeably. When a distinction is necessary, I will explicitly say so.

4.1 Motivation for Synchronizations

When you run multiple processes and threads concurrently, many times you need to enforce some order of executions among the threads or processes. For example, consider the Java program in Figure 4.1. In main, four threads are created one by one, from HelloThread(1) to HelloThread(4). Then each thread is started using the `start()` function. The `run()` method prints out messages with the thread id and the number of times the thread looped.

```
import java.io.*;
import java.lang.*;
import java.util.*;

class MyThreadExampleOneV {
public static void main(String[] args) {
  new HelloThread(1).start();
  new HelloThread(2).start();
  new HelloThread(3).start();
  new HelloThread(4).start();
  System.out.println("Global Variable =
                        " + Global.Variable);
  }
}
class Global {
        public static int Variable = -100;
}
class HelloThread extends Thread {
 int threadID;
 HelloThread(int ID) {
   this.threadID = ID;
 }
 public void run() {
   Global.Variable = this.threadID;
   System.out.println("Thread " + this.threadID +
   " has written " + this.threadID +
   " to Global.Variable");
 } // run
} // end Thread
```

Figure 4.1: A multi-threaded Java program that needs synchronization

First, it is important to note that there are five threads total: the main thread, HelloThread(1), HelloThread(2), HelloThread(3), and HelloThread(4). A naive observation of the program will tell us that we will have the sequence of messages printed out by HelloThread(1) to HelloThread(4) in that order. However, the order of messages will not be guaranteed, which means that there is no specific order of execution of threads defined by the program or OS. One run of the program is shown in Figure 4.2. As shown in the figure, after Thread 1 printed out its message, the main thread printed out the value of the global variable = 2. The programmer most likely in-

tended to print out the message by the main program as the last message. What happened? This is due to how the scheduler chooses the next process to run and the fact that interrupts can occur at any given time at any frequency. Let's try to understand this a bit more in detail.

There can be interrupts generated by various devices during any time, causing the currently running thread to be blocked. If the current thread makes an I/O call, it will be blocked and there is no fixed amount of time that the I/O will take. Therefore, the order of execution among the thread is not deterministic. There can also be other threads belonging to other programs running at anytime. These threads can also run at any time by the scheduler. One fundamental thing to remember is that unless *managed* with special commands, the current process/thread can be interrupted at anytime at any place in the source program.

If the programmer wants threads (or processes) to run in some specific order and synchronized in some way, they need to address this in the implementation of the program. They cannot change the behavior of the scheduler directly. If the programmer does not want a certain thread, say t1, to run until a certain event happen, they must make sure that t1 is not in the ready queue until that event happens. Unless the event happens, t1 is not ready to run; so it should not be in the ready queue. This means the programming language must support such a command. The idea is similar to the `waitpid()` call in Chapter 2 but it is a bit more involved. Read on.

```
Thread 1 has written 1 to Global.Variable
Global Variable = 2
Thread 2 has written 2 to Global.Variable
Thread 3 has written 3 to Global.Variable
Thread 4 has written 4 to Global.Variable
```

Figure 4.2: One possible result of the program in Figure 4.1

4.2 Motivation for Mutual Exclusions

In the program in Figure 4.3, there are three threads that share a global integer array called `Global.buffer[9]` and an index to the array, `index`. Each thread will iterate through its for-loop in the run method three times and write its own id to the buffer array element indexed by the current index value. Notice that at any given time, the currently running thread

can be interrupted and another thread can run because no synchronization commands are used by the programmer.

```
import java.io.*;
import java.lang.*;
import java.util.*;

class MyThreadExampleMutex{
public static void main(String[] args) {
 new HelloThread(1).start();
 new HelloThread(2).start();
 new HelloThread(3).start();
 System.out.print("Global.buffer Content = ");
 for (int i = 0; i < 9; i++) {
  System.out.print(Global.buffer[i] + "; ");
  }}
}
class Global {
 public static int[] buffer = new int[15];
 public static int index = 0;
}
class HelloThread extends Thread {
 int threadID;
 HelloThread(int ID) {
  this.threadID = ID;
 }
 public void run() {
  for (int i = 0; i < 3; i++) {
   Global.buffer[Global.index] = this.threadID;
   Global.index++;
  }
} // run
} // end Thread
```

Figure 4.3: A multi-threaded Java program that shares an array that requires mutual exclusion facilitations. This is an application program that runs on a Java Virtual Machine.

Figure 4.4 shows three possible outcomes, among many others, of the program in Figure 4.3. So what if we want the result to be always "`Global.buffer Content = 1; 1; 1; 2; 2; 2; 3; 3; 3;`"? The programer must use some special commands to ensure this. There are several ways to achieve this

goal. In particular. we will learn synchronization and mutual exclusion using *semaphores* in the next section.

Synchronization ensures a certain order of execution and mutual exclusion makes sure that the shared data is protected. Notice Result3 in Figure 4.4. Because some values are overwritten, unlike in Result1 and Result2, which have exactly three 1s, 2s, and 3s, in Result3, there is only one 1 and two 0s. This is because the index variable is increased multiple times before being used for the assignment statement. Therefore, `Global.buffer[7]` and `Global.buffer[8]` are not written by any threads, consequently have the initial value 0. In other words, say the index variable is 6, right before the index variable is incremented, the current thread, say t1, gets interrupted. Then thread t2 runs and executes `Global.buffer[Global.index] = this.threadID;` and `Global.index++`. Now the index variable is 7. Then t2 is also interrupted. Now t1 starts to run again from the place where it left off, which is right after `Global.buffer[Global.index] = this.threadID;`. In other words, t1 runs `Global.index++`, which makes the index value 8. Notice in this scenario, `Global.buffer[7]` is not used and therefore it retains the initial value 0. The same thing can happen to `Global.buffer[8]`. This problem is known as *race condition.* The race condition describes the situation in which multiple processes or threads are racing to execute certain program instructions that can cause unpredictable results depending on the order of executions. There must be a way to prevent the statement `Global.index++` from being executed again before the incremented index value is used properly for `Global.buffer[Global.index] = this.threadID;`. We discuss this in the next section.

```
Result1:
   Global.buffer Content = 1; 1; 1; 3; 3; 3; 2; 2; 2;
Result2:
   Global.buffer Content = 3; 3; 3; 2; 2; 2; 1; 1; 1;
Result3:
   Global.buffer Content = 3; 3; 1; 3; 2; 2; 2; 0; 0;
```

Figure 4.4: Three of many possible results of the program in Figure 4.3.

4.3 Solutions to Mutual Exclusions and Synchronizations

We now define *mutual exclusion* more formally. Mutual exclusion ensures that no two threads are within the corresponding *critical sections* at the same time. I know, this is a rather cryptic description. Let's clarify this by defining what *corresponding critical sections* are first. Consider the following program in Figure 4.5.

```
object HelloThread(1) {
 int threadID; this.threadID = 1;
 void run() {
  Global.buffer2[Global.index2] = this.threadID;
  Global.index2++;
  for (int i = 0; i < 3; i++) {
   Global.buffer[Global.index] = this.threadID;
   Global.index++;
  }
} // run
} // end Thread(1)

object HelloThread(2) {
 int threadID; this.threadID = 2;
 void run() {
  for (int i = 0; i < 3; i++) {
   Global.buffer[Global.index] = this.threadID;
   Global.index++;
  }
  Global.buffer2[Global.index2] = this.threadID;
  Global.index2++;
  } // run
} // end Thread(2)
```

Figure 4.5: Corresponding Critical Sections. Two pairs of critical sections. The underlined code part in HelloThread(1) and HelloThread(2) are matching critical sections; also the grayed part in HelloThread(1) and HelloThread(2) are another matching critical sections.

There are two sets of critical sections. The codes that are underlined are matching critical sections and the codes that are highlighted are matching critical sections of their own. This means that when HelloThread(1) is executing its underlined code part, HelloThread(2) must not execute its underlined code part, vice versa. The same is true for the code parts that are highlighted in gray. It is completely fine, for example, that HelloThread(1) executes the underlined part and HelloThread(2) executes its gray part, concurrently; this is because there is no shared data between the underlined code part and the gray code part. We showed an example with two threads but there can be any number of threads sharing any data structures. The decision is absolutely up to the application programmer.

As you may have guessed it, the term *critical sections* can be defined as *the part of program code that accesses shared resources, such as data, variables, data structures, and devices.* Remember, a critical section is not a shared resource but the program code that accesses it!

4.3.1 Four Conditions for Mutual Exclusion

There are four conditions for mutual exclusions. Any solution to the mutual exclusion problem must satisfy the following four conditions. The conditions are the same for mutual exclusion of threads.

1. No two processes are simultaneously in the corresponding critical section
2. No assumptions are made about speeds or numbers of CPUs available
3. No process running outside a critical section may block another process that wants to enter a corresponding critical section
4. No process must wait forever to enter its critical section

In fact, these four conditions are very tightly related to each other and all of these conditions must be satisfied by a solution to mutual exclusion. We discuss solutions in Section 4.3.2.

4.3.2 Some Attempted Solutions

In earlier days of computing when multiprogramming techniques were just introduced, many researchers identified the synchronization and mutual exclusion problems. However, they didn't know how to solve the problem correctly. When a paper was published with a potential solution, another

researcher would find a counter example against that solution that would violate one of the four conditions of mutual exclusion. Numerous suggestions have been made. We will discuss some failed attempts first.

Observe Figure 4.5 carefully. Considering the first condition of mutual exclusion, one way to ensure mutual exclusion is to prevent multiple processes from executing the corresponding critical sections simultaneously. We can *protect* each critical section with some kind of entry and exit conditions. This idea is depicted in Figure 4.6. How do we design these two functions: `entry-conditon` and `exit-condition`?

```
object thread1 {                    object thread2 {
 some code                           some code
 void run() {                        void run() {
  non-CS code                         non-CS code
  entry-condition();                  entry-condition();
  Critical Section                    Critical Section
  exit-codition();                    exit-codition();
   non-CS code                        non-CS code
  } // run                           } // run
} // end thread1                    } // end thread2
```

Figure 4.6: Protecting the matching critical sections for mutual exclusions

A funny example (or rather an analogy) I use to explain this idea is the *bathroom problem*. Imagine you are living with a roommate in an apartment. Somehow you and your roommate do not talk to each other or see each other.[1] Now there is only one bathroom available and you and your roommate must not enter the bathroom at the same time! Mutual exclusion! How would you and your roommate accomplish this?

Turning off Interrupts

The origin of the race condition problem is that an interrupt can occur at any time. If we turn off all interrupts before entering a critical section and turning them on again as we exit the critical section, all codes with the critical section can execute *atomically*. Atomically executing a set of program

[1] The reason can be anything you can imagine. You had a big fight with your roommate so she is transparent to you or you and your roommate do not speak the same language at all. Just come up with your own reason. The reason is not important.

commands means that all commands in the set are executed without any interruption.

```
void entry_condition() { turn_off_interrupts(); }
void exit_condition() { turn_on_interrupts(); }
```

Figure 4.7: The `entry_condition()` and the `exit_condition()` functions turning off and on interrupts to solve race condition.

At first glance, this 'solution' may look fine. However, it is not a practical solution. Why? Note that turning interrupts on and off are privileged instructions that should only be allowed to OS; in fact, to the one of the lowest-levels within OS. Remember that the programs that will use these two functions are mostly application programs. We are giving too much privilege to application programmers. OS designers should not trust application programmers in the sense that application programmers may make mistakes or intentionally write malicious programs. OS must protect the system. What if an application program turns off interrupts by calling `entry_condition()` but forgot to call `exit_condition()`? If this computer happen to be a server machine then no one else can use the computer because the interrupt mechanism would not work after that.

Use a Flag

Going back to the roommate situation; you and your roommate came up with the idea to use a sign on the bathroom door. The sign "0" indicates the bathroom is empty and "1" indicates it is being used. This approach looks promising. After all, humans could do this and it would work for us. Unfortunately, this approach will let two processes (or threads for that matter) within a matching critical sections at the same time. In our bathroom analogy, you and your roommate will find each other in the bathroom! We don't want that happen.

```
boolean global variable being_used = false;
void entry_condition() {
 while (being_used != false) ; // loop until 0
 being_used = true;   // set it to 1 before entering
}
void exit_condition() { being_used = false; }
```

Figure 4.8: The `entry_condition()` and the `exit_condition()` functions using a boolean variable (a flag) to solve race condition.

Let's see how this violation of mutual exclusion happens. In order to understand this, you need to put yourself in the process's (or thread's) shoes, if these things had shoes, of course.

Consider the programs in Figure 4.6 with two threads, `thread1` and `thread2` having a matching set of critical sections. Consider a scenario in which `thread1` starts to run first and call `entry_condtion()`. Inside of the function, right after checking the `while-loop` statement condition, which evaluates to false, it will break out of the loop. Just before the next statement `being_used = true;` is executed, a timer interrupt occurs and `thread2` runs. `Thread2` now runs its `entry_condtion()`. Because `being_used` is still false, `thread2` will also break out of the loop and execute further to `being_used = true;` and enter the critical section. While `thread2` is still executing codes in the critical section,[2] another timer interrupt occurs and now `thread1` is restarted. When `thread1` is restarted, it will not go back and execute the while-loop again. It will first execute `being_used = true;` because that is the next command to be executed. Now `thread1` will also enter its critical section. Both threads will now execute the matching critical sections simultaneously. This can cause the race condition and a non-deterministic result, as we described in Figure 4.3.

The problem with this approach is that the global boolean variable `being_used` is also a shared variable between the threads. This means it also has to be protected. Any code that accesses this variable is a critical section. That is the `entry_condition()` and `exit_condition()` functions are also critical sections that needs to be protected using another set of `entry_condition()` and `exit_condition()` functions. This problem will still repeat even if we have another set of these two functions. Ad infini-

[2]Notice that there is no assumption about how many lines of codes in the critical section. Therefore, it can take a long time.

tum.

Use a Flag a Bit Differently

Now let's use a boolean variable a bit differently. Instead of checking if the bathroom is empty or not, the roommates decided to check whose turn it is to use the bathroom. See the functions in Figure 4.9. It has a global variable turn initialized to 0. When a process wants to enter a critical section, it executes `entry_condition(process id)` with its id as the parameter. For example `thread1` will call the function as `entry_condition(1)` and `thread2` will call the function as `entry_condition(2)`. The same is true for `exit_condition(process id)`.

```
boolean global variable turn = 1;
void entry_condition(process id) {
 while (turn != id) ; // loop until matches its id
 turn = id;   // set it to 2 before entering
}
void exit_condition(process id) {
 if (id == 1) turn = 2; else turn = 1;
}
```

Figure 4.9: The `entry_condition()` and the `exit_condition()` functions using a turn flag variable to solve race condition. This uses a *busy waiting*– i.e., the loop.

This approach will actually make sure no two threads are in the matching critical sections at the same time. However, it will strictly enforce an alternating sequence of critical sections. This means the roommates (say Jack and Sally) are taking turns of using the bathroom. What if Jack wants to go to the bathroom twice in a row? Jack can't do that unless Sally uses the bathroom. Jack must make Sally—who doesn't need to go to bathroom—to use the bathroom in order for him to use it. This violates condition 2, 3, and 4 of mutual exclusion described in Section 4.3.1. Therefore, it is not a solution.

4.3.3 Some Solutions with Busy Waiting

There are some solutions that use busy waiting that works well. Again busy waiting means a process wanting to go into a critical section is in a while-loop

checking the condition repeatedly until the shared resources are released by the process that is currently using its critical section. Peterson's solution is a good example. We will not discuss the solution in this book. However, it is important to understand that such a solution is not used anymore because busy-waiting is considered bad for the following reasons:

- Busy-waiting wastes CPU cycles: In order to check the condition repeatedly, CPU must execute the while-loop.
- Busy-waiting can cause the priority inversion problem: Consider a scenario in which a lower priority process is currently running it's critical section. In the middle of executing the critical section, a higher priority process has been just introduced to the system and therefore it is inserted in the ready queue. The scheduler will now start running the higher priority process. Coincidentally, the higher priority process wants to enter a corresponding critical section that the lower priority process was executing. Because the lower priority process is still in its critical section, the higher priority process must wait in the while-loop. Alas! the scheduler will never let the lower priority process to run and finish its critical section because it will always give the CPU to the higher priority process. Now it is a situation in which a higher priority process is blocked from making progress because a lower priority job cannot continue. We call such a situation generally the *priority inversion problem.*
- Busy-waiting is bad for memory hardware: This is a rather old archaic problem. Before transistors were invented, main memory was constructed with magnets and coils. It was called *core memory.* Magnets can loose their polarity if used excessively. Checking the conditional within the while-loop repeatedly reads the same memory core over and over again, which can cause the magnets assigned to the conditional variable within the while-loop to be over used and loose their polarities.

For these reasons, busy waiting solutions are rarely used. At a high level, a better solution would be to let a waiting process to be blocked until the other process is done with its critical section. When it is done, the waiting process should be woken up to the ready queue. We discuss a solution using this idea in the next section.

4.3.4 A Better Solution: Semaphore with a Wait Queue

A well-known computer scientist Edsger W. Dijkstra, came up with an idea, possibly inspired by a railroad control system, where two train tracks cross a shared resource. No two trains use the cross section at any time for an obvious reason. A signaling system works well to ensure that. He introduced the concept of semaphores that defines two functions called P() and V(). The term semaphores is a dutch word, Proberen and Verhogen are wait and signal in dutch (more or less).

```
class semaphore (int init_value) {
 int count = init_value;
 List queue;     // a queue of processes
 void P();
 void V();
}
void P() {
    turn-off(interrupts);
    count = count -1;
    if (count < 0) {
       block (this.proces);
       enqueue (this.process);
    }
     turn-on(interrupts);
}
void V() {
    turn-off(interrupts);
    count = count +1;
    if (count < = 0) {
    p = dequeue();  // remove a process in queue
    insert_to_ready_queue(p);
    }
    turn-on(interrupts);
}
```

Figure 4.10: The semaphore solution to mutual exclusion

There are several different ways to implement semaphores.[3] Figure 4.10

[3]In some literature, the count variable in semaphores are to be non-negative integer. The implementation we present is also widely used. I believe this implementation is more intuitive and simpler to understand.

defines the semaphore class and two member functions `P()` and `V()`. `P()` and `V()` are to be used in place of `entry_condition()` and `exit_condition()`, respectively.[4] Some literatures use `wait()` for `P()` and `signal()` for `V()`. I have also seen people using `down()` and `up()`, respectively. These variations will all make sense once you understand the mechanism of semaphore.

Notice that semaphore operations `P()` and `V()` are atomic operations. There can't be interrupts while executing the functions. One may ask, wait a minutes, haven't we said that we can't use this method because it gives too much privilege to the application programmers? Notice that the semaphore is at the kernel level implemented by OS developers; application programers are just to use the implementations. Therefore, the kernel (i.e., OS) programmers must ensure the interrupts are properly turned off and on at the right times. If we can't trust the OS designers and implementers, then what can we do? You also should notice that in `P()` if`count < 0`, the process calling `P()` gets blocked until the counter is incremented by a `V()`. When the process is blocked, next process in the ready queue must be scheduled to run. Unseen in the semaphore implementation, interrupts are turned back on by the OS.

The semaphore class consists of an integer counter, called `counter` and a queue of processes (or threads). The counter is initialized with an appropriate value depending on the purpose as we discuss later. For mutual exclusion, the counter variable is usually initialized to 1.

[4] We can continue to use our function names `entry_condition()` and `exit_condition()` but to be consistent with other literature, we use `P()` and `P()`.

```
global s = new semaphore(1);

object thread1 {                    object thread2 {
 some code                           some code
 void run() {                        void run() {
  non-CS code                         non-CS code
  s.P();                              s.P();
  Critical Section                    Critical Section
  s.V();                              s.V();
   non-CS code                        non-CS code
  } // run                            } // run
} // end thread1                     } // end thread2
```

Figure 4.11: Protecting the matching critical sections for mutual exclusions using semaphores

Figure 4.11 shows mutual exclusion using semaphores. Let's consider the same scenario that violated mutual exclusion when using previous attempts with semaphores. That is a scenario in which `thread1` starts to run first and call `s.P()`. Because the counter is 1, it is decremented to 0 and `thread1` enters the critical section. While `thread1` is in the critical section, a timer interrupt occurs and `thread2` starts to run. `thread2` now executes its own `s.P()`. Because the counter is zero, count will be -1 and `thread2` will be blocked and inserted into the semaphore's `s.queue`. See Figure 4.12.

Now the scheduler chooses to run `thread1` again and `thread1` finishes its critical section and calls `s.V()`. This will increment count to 0 first. Then, because `if (count <=0)` is true, `p = dequeue()` will dequeue `thread2` from `s.queue` and insert it to the ready queue. Once `thread2` is in the ready queue, the scheduler will eventually choose to run it. When `thread2` runs again, it will start running from the code right after `enqueue(this.process)`, therefore, it comes out of `s.P()` and goes into the critical section. There is no violation of mutual exclusion.

As you analyze multiple processes or threads using semaphores, it is important to follow the values of various semaphores counters and contents of the queues. When someone says "semaphores," you should immediately be reminded of a counter variable and a queue!

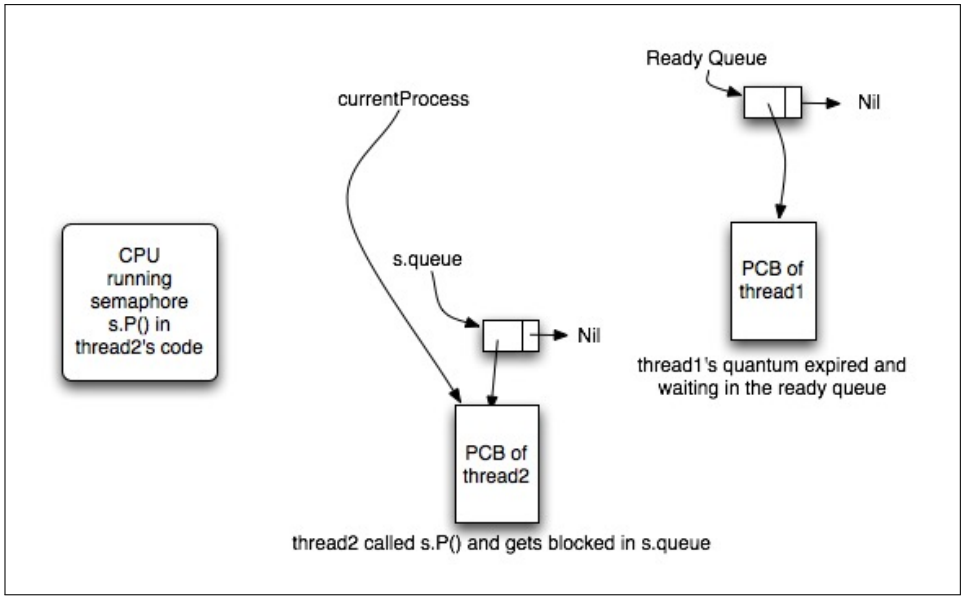


Figure 4.12: A snapshot of OS status after `thread2` gets blocked while executing `s.P()`

When you have multiple shared resources to be protected for mutual exclusion, you should use multiple semaphore objects as in Figure 4.14. In the figure, we create two semaphore objects s1 and s2. s1 protects the critical sections underlined and s2 protects the grayed critical sections. That is, semaphore's P() and V() are just programming commands that programmers can use to enforce synchronizations and mutual exclusion. It is the programmers job to figure out how to use semaphore calls.

We discussed how semaphores can be used for synchronizations and we discussed mutual exclusion. Now it is time to discuss synchronizations. Figure 4.1 showed an example of processes that need synchronizations. Go back to the figure and refresh your knowledge.

4.3.5 Some Well-known Classical Synchronization and Mutual Exclusion Problems

In early days of multiprogramming research, computer scientists liked to come up with analogous examples in real life to described synchronization and mutual exclusion problems. We have already discussed two: the mutual

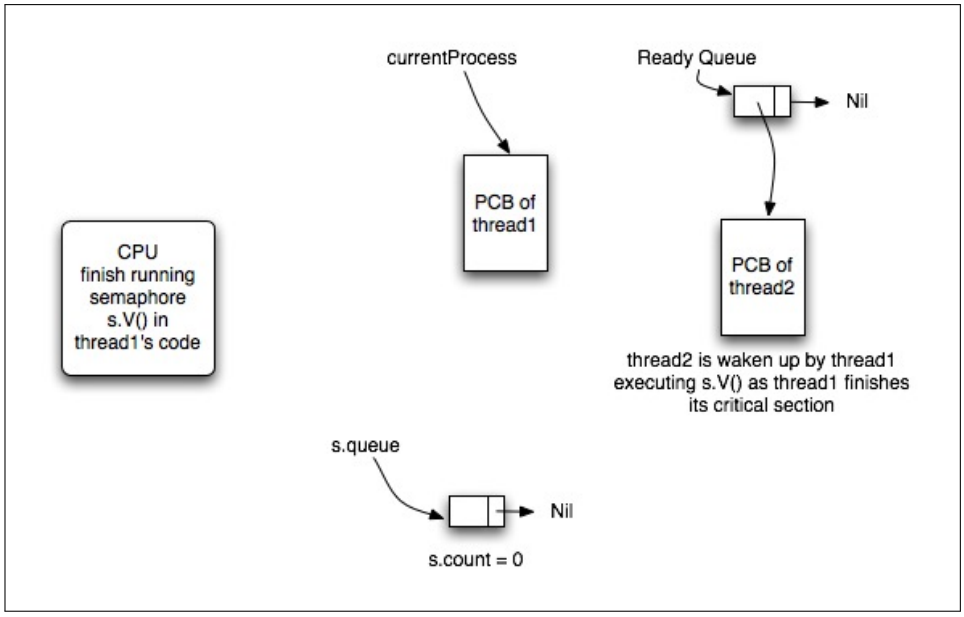


Figure 4.13: A snapshot of OS status as `thread1` finish executing `s.V()` waking up `thread2` from `s.queue` to the ready queue.

exclusion on a cross-section of two train tracks as shared resources and the bathroom example.

We will discuss more of such problems and study synchronizations and mutual exclusions using semaphore.

The Producer and Consumer Problem

The producer and the consumer problem can be described with two processes one is called the producer and the other the consumer. We can consider multiple producers and consumers but the solution to one producer and one consumer problem works well in multiple cases.

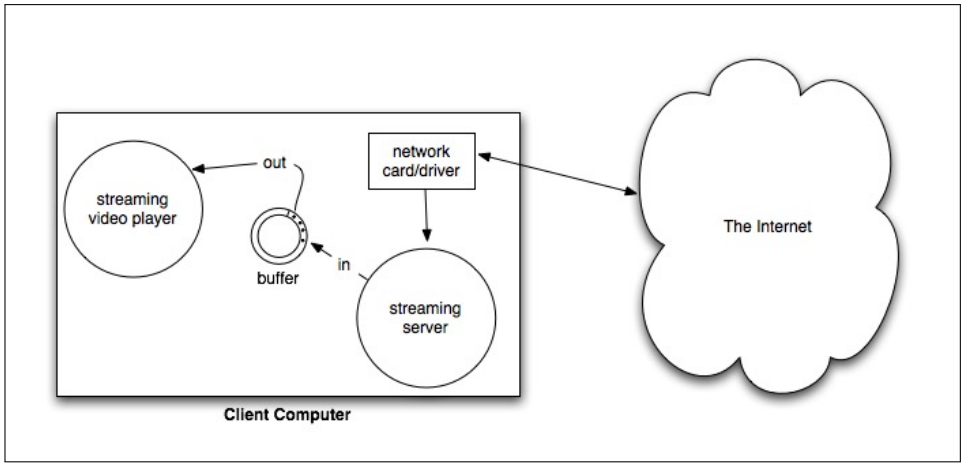


Figure 4.15: An illustration of a streaming video player getting video packets from the Internet and playing a movie.

I will use an example of the producer and consumer problem out of an Internet streaming video (or music) program. Figure 4.15 shows such an example. Because the video data packets are accessed through the Internet from a web server far away, the network connection speed will most likely vary during the video play. In other words, there will be times that packets are arriving too fast from the remote server so that unplayed packets needs to be saved in a buffer. It is also possible that the network connection speed becomes slow so that packets are not arriving fast enough. For this situation, the streaming video player should wait until next available packets arrive if the buffer is empty. Basically the streaming video player is the consumer (of the packets) and the the streaming server is the producer (of the packets). Notice that the client computer does not have a control over the remote web server's speed nor the network speed. The consumer must wait if there is no packets available and the producer must not send more packets if the buffer is full. We will discuss how to implement this using semaphores.

```
final int BUF_SIZE = 100;
packet buffer = new packet[BUF_SIZE];
int numItems = 0 // number of items in buffer
queue = new wait_queue();   // assume this is under OS

object streaming_server() {
 void run() {
  while (true) {
   vpacket = produce_packet();
   if (numItems == BUF_SIZE) wait(queue);
   insert_packet(vpacket, buffer);
   numItems++;
   if (numItems == 1) wake_up(videoplayer);
   }
  } // run
} // end streaming_server

object videoplayer() {
 some code
 void run() {
  while (true) {
   if (numItems == 0) wait(queue);
   vpacket = get_packet(buffer);
   numItems--;
   if (numItems == MAX-1) wake_up(streaming_server);
   diplayvideo(vpacket);
   }
  } // run
} // end videoplayer
```

Figure 4.16: A streaming video player process code that has problems

Figure 4.16 shows an attempt to solve the problem by directly translating the description of the behaviors of `streaming_server()` and `video_player()`. Unfortunately, this attempt would fail. Notice that the buffer and numItems are shared resources between the two threads and there are matching critical sections that access these resources in the threads. This means that we need to protect the critical sections using semaphores. We also declare a wait queue for the threads to sleep. However, the queue must be implemented in the OS not in the application program. Remember that the semaphore class defines a queue for a thread to wait on. Then why don't we use semaphore

queues to have threads to wait on?

```
final int BUF_SIZE = 100;
packet buffer = new packet[BUF_SIZE];
global mutex = new semaphore(1);
global numPacketsInBuffer = new semaphore(0);
global numAvailBufferSlots = new semaphore(BUF_SIZE);

object streaming_server() {
 void run() {
  while (true) {
   vpacket = produce_packet();
   numAvailBufferSlots.P();
   mutex.P();
    insert_packet(vpacket, buffer);
   mutex.V();
   numPacketsInBuffer.V(); // one more packet available
   }
  } // run
} // end streaming_server

object videoplayer() {
 some code
 void run() {
  while (true) {
   numPacketsInBuffer.P();
   mutex.P();
   vpacket = get_packet(buffer);
   mutex.V();
   numAvailBufferSlots.V();
   diplayvideo(vpacket);
   }
  } // run
} // end videoplayer
```

Figure 4.17: A streaming video player code that has problems

Figure 4.17 is a solution to the streaming video problem using semaphores.

`streaming_server()` first checks if there is at least one available buffer slot by calling `numAvailBufferSlots.P()`; if so, call `insert_packet(vpacket, buffer)`. For example, if the `streaming_server()` runs first in the beginning, there is 100 available buffer slots. Executing `numAvailBufferSlots.P()`

will decrement the `count` variable in `numAvailBufferSlot` semaphore by 1. Now it will be 99. To see this, review the code for semaphore `P()` operation in Figure 4.10. Of course the `insert_packet(vpacket, buffer)` call must be protected by mutex semaphore operations. That is done by the pair of `mutex.P()` and `mutex.V()`. After the insertion of a packet, the server calls `numPacketsInBuffer.V()` to increment the number of packets in the buffer by 1. Also if `videoplayer` has been waiting on the numPacketsInBuffer's semaphore queue, this `V()` operation will wake it up and put it in the ready queue.

The `videoplayer` thread works similarly. It will call `numPacketsInBuffer.P()` to check if there is at least one packets in the buffer to consume to display, if not `videoplayer` will be blocked in the numPacketsInBuffer's semaphore queue. Otherwise, it will continue to get a packet from the buffer. Of course the `vpacket = get_packet(buffer)` call must be protected by a mutex semaphore because `buffer` is a shared resource. The `videoplayer` now will call `numAvailBufferSlots.V()` to increment the number of available buffer slots by 1 because `videoplayer` just extracted a packet from the buffer so that one more slot is available.

An Illustrated Example

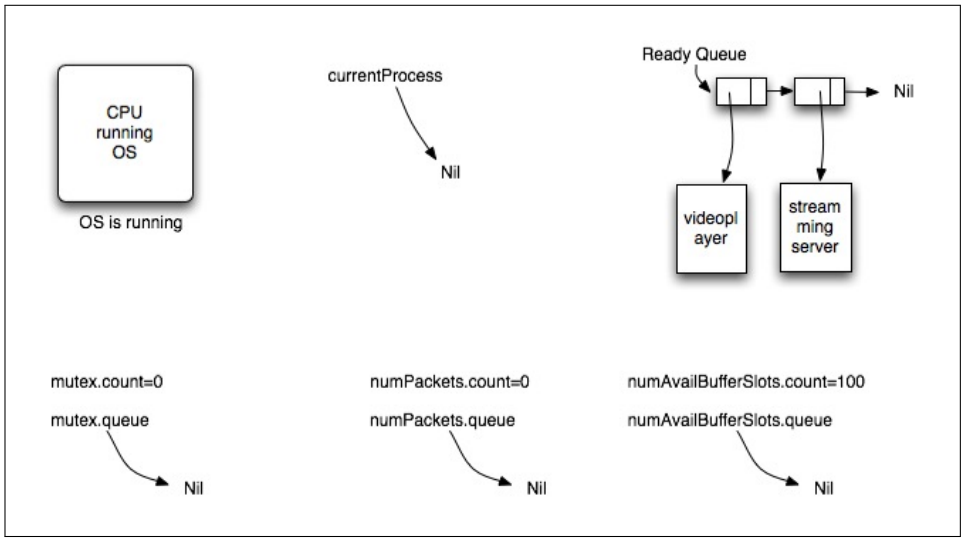


Figure 4.18: An illustration of `videoplayer` and `streaming_server` problem using semaphores. An initial view as in the example explained.

Assume that Figure 4.18 shows the initial internals of OS before the two threads runs on the CPU but are ready in the ready queue. Notice that `numPacketsInBuffer.count = 0` and `numAvailBufferSlots.count = 100` initially. The queues for both semaphores are empty intially. Also `mutex.count = 1` and its queue is empty. Usage of mutex semaphores is relatively easier than synchronization semaphores. You just need to make sure there are *logically* matching `P()` and `V()` calls wrapping around the shared resource to be protected. Caution: you can't just count number of `P()` and `V()` calls in the source code. Why not? This is because, for example, a `P()` call can be conditionally made within an `if-then` clause. That means depending on the truth value of the `if-condition`, a matching `V()` may or may not be needed.

Now let's focus back on the synchronization semaphores: i.e., `numPacketsInBuffer` and `numAvailBufferSolts`. Note that in Figure 4.18, `videoplayer` is first in the ready queue. The scheduler runs it and calls `numPacketsInBuffer.P()` in the while-loop. Since `numPacketsInBuffer.count = 0`, `videoplayer` will be blocked in its semaphore queue after decreasing `numPacketsInBuffer.count = -1`. Since `videoplayer` cannot continue (because there is no packet to consume), the scheduler will start running `streaming_server`.

Figure 4.19 shows the situation in which `streaming_server` just finished `vpacket = produce_packet()` before running `numAvailBufferSlots.P()`. Now `streaming_server` continues and executes `numAvailBufferSlots.P()`. Note `numAvailBufferSlots.count` is 100, when `streaming_server` finishes `numAvailBufferSlots.P()` it will be 99. It will continue to execute `mutex.P()`, making `mutex.count` to 0, execute `insert_packet(vpacket, buffer)` and then `mutex.V()`, making `mutex.count` to 1 again as it comes out of the critical section. When `streaming_server` continues to execute `numPacketsInBuffer.V()`, the `V()` call will set `numPacketsInBuffer.count = 0` and then wakes up `videoplayer` from `numPacketsInBuffer.queue` and insert it in the ready queue. See Figure 4.20 for the snapshot of the situation.

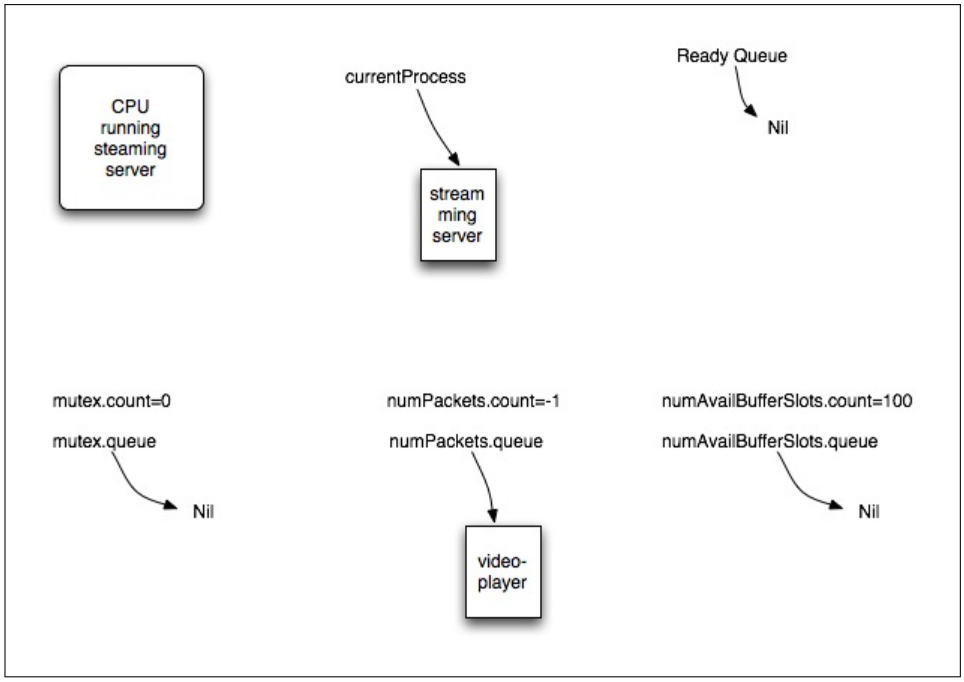


Figure 4.19: After the `videoplayer` process gets blocked and the `streaming_server` process runs

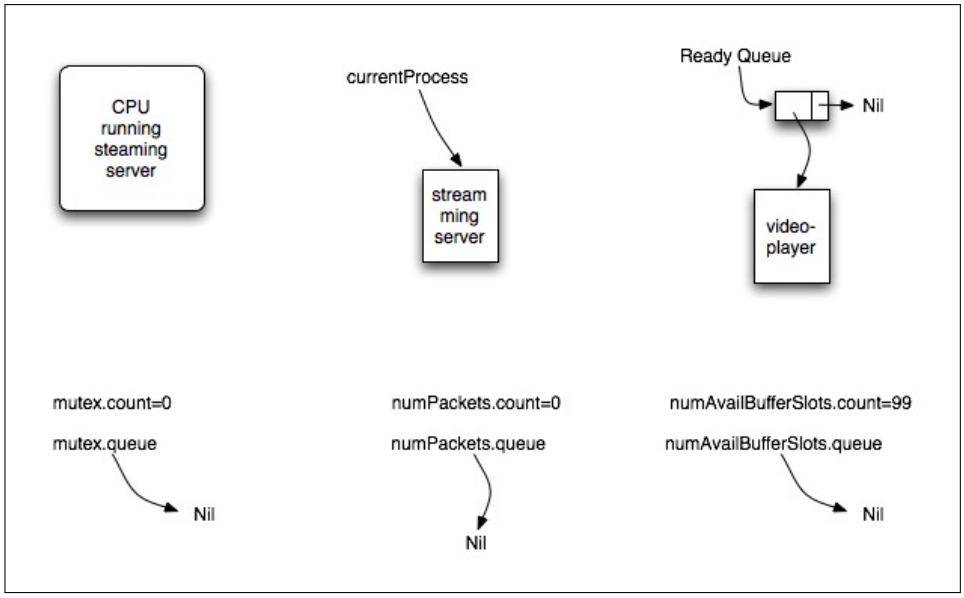


Figure 4.20: The `videoplayer` process gets waken up and inserted back in the ready queue by the `streaming_server` process executing `numPacketsInBuffer.V()`.

Continuing on with the example, let's say somehow the `streaming_server` process runs for 99 iterations of the while-loop without letting the `videoplayer` process run. Eventually the buffer will be full with video packets because `videoplayer` did not have a chance to consume the packets in the buffer. After the buffer is full, `streaming_server` executes `numAvailBufferSlots.P()`. Because `numAvailBufferSlots.count` is 0, `numAvailBufferSlots.count` becomes -1, and the thread will get blocked in `numAvailBufferSlots.queue`. Figure 4.21 shows this situation.

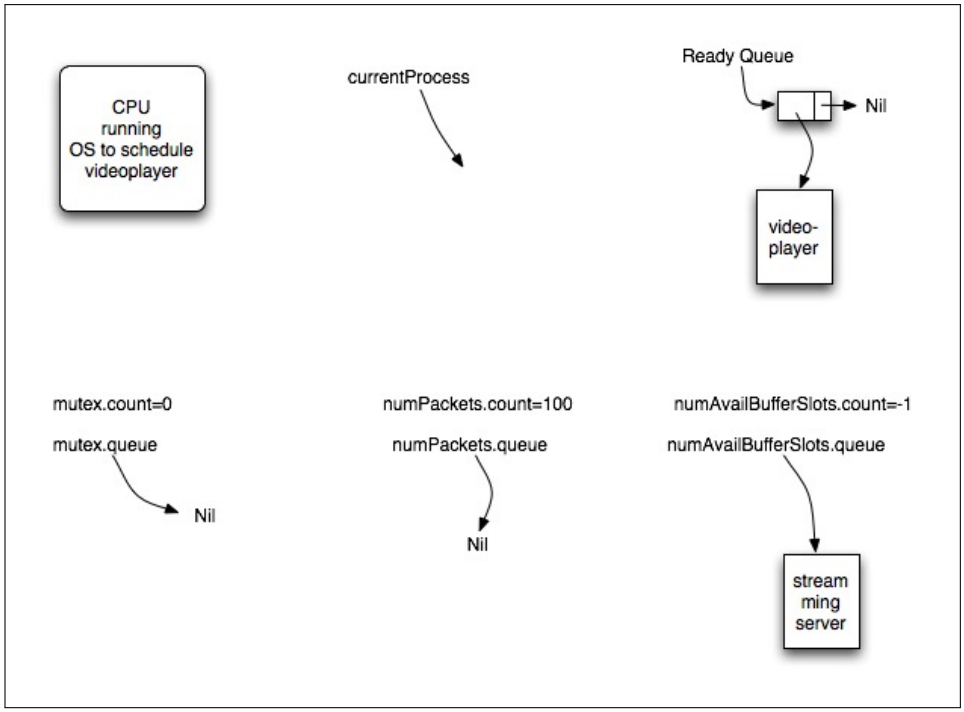


Figure 4.21: The `streaming_server` process gets blocked after executing `numAvailBufferSlots.P()` because the buffer is full, therefore cannot continue. The `videoplayer` process is next to run.

Now `videoplayer` resumes running and executes `mutex.P(), vpacket = get_packet(buffer), mutex.V()`, and `numAvailBufferSlots.V()`. In `numAvailBufferSlots.V()`, `numAvailBufferSlots.count` is incremented to 0 and `streaming_server` will be woken up and inserted back in the ready queue. Figure 4.22 shows the snapshot of the situation.

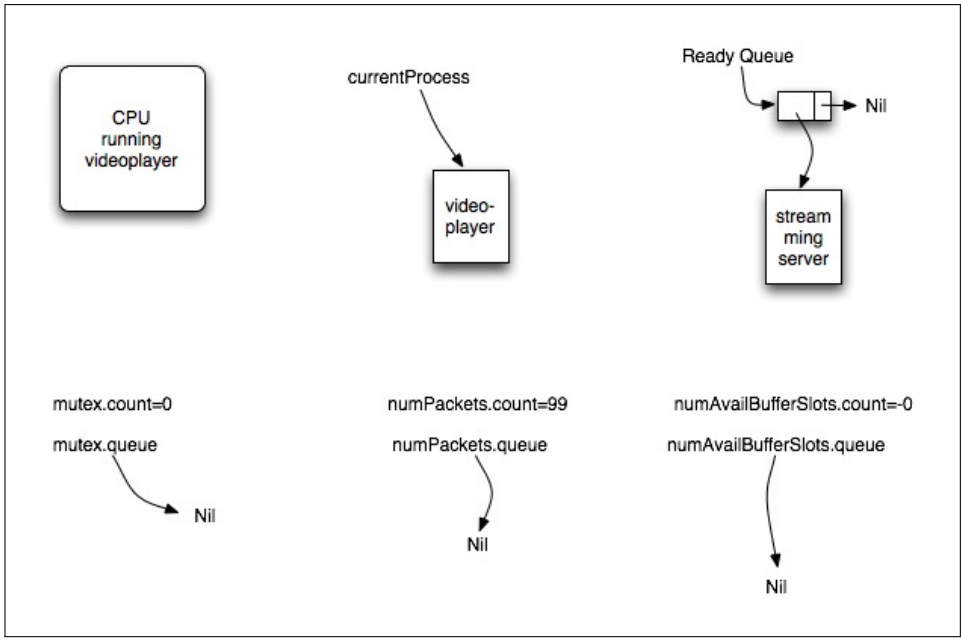


Figure 4.22: The `streaming_server` process gets waken up by the `videoplayer` process executing `numAvailBufferSlots.V()`.

It is important to understand these snapshots of the example; furthermore, you need to be able to produce a snapshot for any given scenario of the two threads running.

4.3.6 Other Classical Synchronization Problems and Problems with Semaphores

There are many so called synchronization problems that exist. They are known as the dinning philosophers problem, readers and writers problem, sleeping barbers problem, Santa Clause problem. They are all solvable using semaphores. Interested readers should consult other materials to study these problems by referencing the literatures presented at the end of the chapter.

Although these problems are solvable using semaphores, semaphores are not considered an excellent choice in modern programming practice. Don't get me wrong, semaphores are still very important and many programmers use them. So you must understand that when semaphores are considered less-than-excellent tools, it is mostly coming from a software engineering perspective. In other words, semaphores are difficult to use for programmers

because they must make sure that for every `P()` operation, there must be a logically matching `V()` operation. This may sound easy because some people may think that we can just count the number of `P()` calls and the number of `V()` calls to make sure there are equal numbers of both in the source code. It is not that simple. As explained before, calling these functions can be conditional. In other words, programmers can call a `P()` only if a certain condition is met. Then a matching `V()` may or may not be needed depending on the program flow. Whether the condition is satisfied or not is a run time information and without running the program, we cannot predict the condition.

Researchers have developed other synchronization constructs that are higher-level than semaphores. One such effort is the concept of *monitor*. The monitor is a high-level language construct that can be used by programmers. Any code enclosed within a monitor declaration must be considered as synchronized. There are several implementations of this language construct. In Java, the `synchronized`method implements one version of monitor. Figure 4.23 shows the synchronized version of the previously introduced program in Figure 4.3. Within the `synchronized` constructs, all data structures are protected and the program codes within are considered as a critical section. Notice however, synchronized is a misnomer in some sense, because it does not define the order of executions of threads. It just makes sure that once a thread is in the critical section, no other threads can run the critical section unless the one already in the critical section finishes it.

```
import java.io.*;
import java.lang.*;
import java.util.*;

class MyThreadExampleMutexSynch{
public static void main(String[] args) {
 new HelloThread(1).start();
 new HelloThread(2).start();
 new HelloThread(3).start();
 System.out.print("Global.buffer Content = ");
 for (int i = 0; i < 9; i++) {
  System.out.print(Global.buffer[i] + "; ");
  }
  System.out.print("\n");
}}
class Global {
 public static int[] buffer = new int[15];
 public static int index = 0;
}
class HelloThread extends Thread {
 int threadID;
 HelloThread(int ID) {
  this.threadID = ID;
 }
 public void run() {
  synchronized(this) {
  synchronized(Global.buffer) {
  for (int i = 0; i < 3; i++) {
   Global.buffer[Global.index] = this.threadID;
   Global.index++;
  }}}
} // run
} // end Thread
```

Figure 4.23: A multi-threaded Java program that shares an array that requires mutual exclusion facilitations and using Java's `synchronized` construct to protect shared resources.

Note that the monitor construct is not independent from semaphores. It is just a high-level construct to ease programmers in concurrent programming. The compiler will eventually translate synchronized blocks into semaphores or other lower-level synchronization commands. There are other

synchronization constructs but we will not discuss them in this book.

4.4 Deadlocks

A deadlock is a situation in that multiple threads (or processes) are waiting for certain events to happen but those events can only occur by one or more waiting threads. The concept is best explained in the diagram in Figure 4.24. In the diagram, process1 is waiting for an event that can only be generated by process2 and vice versa. That is process1 is currently using the printer while asking for the cd rom drive and process2 is currently using the cd rom drive yet waiting for the printer. We call this graph the wait-for graph. In this case, both processes are "stuck" and waiting in a waiting queue. The waiting queue can be an I/O queue, a semaphore queue, or any other queue other than the ready queue.

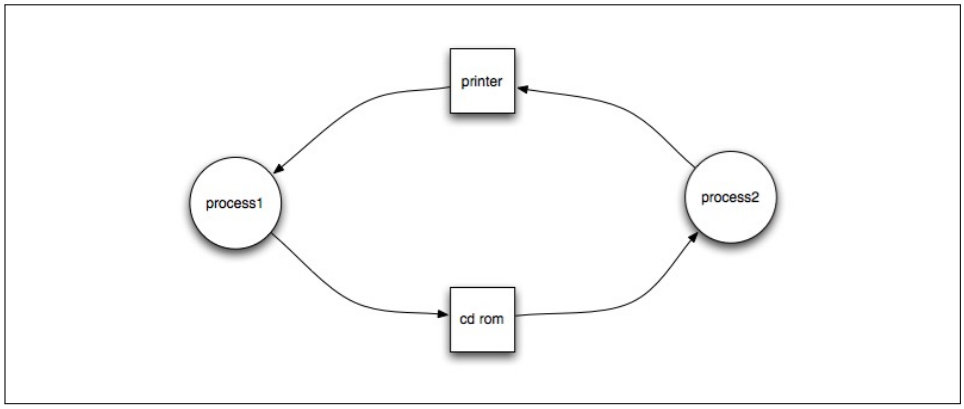


Figure 4.24: A simple deadlock situation involving two processes

Although the example is simple and innocent enough, for a server computer possibly running a large number of processes at any given time, detecting some small number of processes that are stuck among each other is difficult and time consuming. It is time consuming because the detection is done by a subroutine in OS. That means while the detection subroutine runs, processes for user application programs get less amount of computing resources (including the CPU). In many cases, because the system snapshot of the currently running processes are to be analyzed, no other processes (of course including user application processes) can run during a deadlock detection process.

Because of the high overhead, many OSs ignore the issue of deadlock and do not attempt to resolve the potential issue. When you see your computer screen freeze, after all, you can reboot the system. Problem solved! This works if you are using a non-critical system such as an office machine or a personal computer. However, imagine a computer system that controls a nuclear power plant. We can't just simply reboot the system at will. There has to be some way to deal with this.

The issue of deadlock, in fact, has been extensively studied, largely due to the fact that it can be modeled in relatively pure mathematical models. Therefore, many theoreticians have studied this problem using varieties of different mathematical models including graph theory and petri-nets.

There are three ways in general to address deadlock issues in addition to completely ignoring the issues.

- deadlock detection: a deadlock detection approach would let deadlocks occur, but tries to detect deadlocks in some scheduled times. For example, OS can run a deadlock detection subroutine every hour to see if there are processes that are locked in each other. One way is to construct the wait-for graph and traverse the graph to detect a cycle. If a cycle is detected, then the processes within the cycle are in a deadlocked state. One way to break the cycle is that to kill one of the processes within the cycle.

- deadlock avoidance: When a request for certain event to happen (e.g., a request of a printer), OS examines whether granting the request would put OS in an *unsafe* state. An unsafe state is not a deadlocked state but there is a possibility for a deadlock. An analogy to this is a bank lending monies to people. A bank has depositors as well as borrowers. Banks lend monies to borrowers using their depositors money. If all depositors want to withdraw money all at the same time, banks will be bankrupt, for the lack of a better word. So if banks lends more money than the usual withdrawal amount by depositors, they are in an unsafe state but not necessarily bankrupt, because the chances are that not all depositors would withdraw all of their money all at the same time. Dijkstra developed the so-called *bankers algorithm* to achieve deadlock avoidance. Notice that deadlock avoidance is a dynamic algorithm in the sense that the algorithm is always checking whether each resource request is safe or not. On the other hand, deadlock detection is a static algorithm in the sense that when deadlocks occur, the algorithm does not run. Only after the fact, a detection algorithm tries to fix the problem.

- deadlock prevention: deadlock prevention attempts to prevent deadlock from happening all together by establishing a policy. For example, we can give a unique number id to each resource at the OS boot time, say the printer will be id 1 and the cd rom will be id 2. When a process requests a resource, it must request and acquire resources in their increasing id numbers. Therefore, a process will not be able to acquire cd rom drive (id = 2) unless it has already acquired the printer (id = 1).

Exercises

Ex. 4.1 — A round-robin with quantum scheduling allows processes to be switched after the quantum amount is used by the current process, the timer device is a hardware component that can generate an interrupt after one quantum is over. Write a pseudo code for a timer-interrupt handler that can make a context switch after a timer interrupt.

Ex. 4.2 — In implementing semaphore class, we introduced a counter variable. Isn't this variable also shared by multiple processes which needs to be protected as well? To do this, interrupts are disabled before entering the semaphore's P() and V() methods and they are turned back on after. We discuss that turning on and off interrupt is not a good solution to mutual exclusion because it gives too much privilege to users. Why is this ok?

Ex. 4.3 — Consider the situation that there are two processes waiting in the ready queue, P1 and P2. The current process is P0. P0 calls fork() and creates a child process P0C and P0 continue to run without giving up the CPU. What does the ready queue look like at this time?

Ex. 4.4 — Continuing the scenario in Exercise 4.3, P0C calls `waitpid()` so it gets blocked. Now P1 runs. What are the processes in the ready queue. Is P0 in the ready queue? If not, where is it?

Ex. 4.5 — The `yield()` system calls also let the current process give up the CPU as `waitpid()`. Discuss differences between the two.

Ex. 4.6 — Under a Unix environment, type the program in Figure 4.1 and save it as `MyThreadExampleOneV.java`. Compile the program and run it multiple times. Observe the outputs and explain the outputs.

Ex. 4.7 — In Page 134, we read that at any given time, the currently running thread can be interrupted and another thread can run. Explain at least two cases for this situation.

Ex. 4.8 — Under a Unix environment, type the program in Figure 4.3 and save it as `MyThreadExampleMutex.java`. Compile the program and run it multiple times. Observe the outputs. Discuss the difference in outputs between this program and the program in Exercise 4.6.

Ex. 4.9 — Does the program in Figure 4.3 achieve synchronization, meaning that it runs threads in a specific order always? Why not?

Ex. 4.10 — Discuss the differences between mutual exclusion and synchronization.

Ex. 4.11 — Define *critical sections* precisely.

Ex. 4.12 — When programmers write programs with critical sections, do they need to know how many processes will be accessing the critical sections concurrently? Explain.

Ex. 4.13 — The second condition for mutual exclusion is "No assumptions are made about speeds or numbers of CPUs available." (1) Come up with a scenario that would require this condition, and (2) Discuss why different speeds of CPUs and number of CPUs available cause the same scenario as you discuss in (1).

Ex. 4.14 — In this chapter, we discussed a semaphore implementation with a block queue. Implement a semaphore using busy-waiting without a queue.

Ex. 4.15 — Explain a scenario that violates the conditions of mutual exclusion using the code in Figure 4.9.

Ex. 4.16 — Read about the sleeping barber problem on the Internet or in books. Use semaphores to solve the problem.

Ex. 4.17 — Read about the Santa Claus problem on the Internet or in books. Use semaphores to solve the problem.

Ex. 4.18 — Consider a shared memory area SM. There are two processes, Requester and Server, communicating through the share memory SM. At first, Requester performs Action 1 by writing a request, then signals Server, letting it know that the request has been written to SM, Server then performs Action 2 to read the request, Server will process the request and write the result of the request to SM (this is Action 3, then it signals Requester. Requester then reads from SM (Action 4). Requester and Server do this "dance" infinitely. Write pseudo-codes for Requester and Server using semaphores to synchronize these activities in that order. Make sure to state the initialization values of the semaphores used.

Ex. 4.19 — Recall the pseudo-code in Figure 4.17. If `videoplayer()` runs first before `streaming_server` produces its first item, what happens to `videoplayer()` ? What are the semaphore counter variable values and what

do semaphore queues look like at this time (for both `numPacketsInBuffer` and `numAvailBufferSlots`)?

Ex. 4.20 — Give a scenario of the Priority Inversion problem to happen in programs using the busy-waiting solution such as the Petersons algorithm. (Assume that there are two processes P1 and P2, P1s priority is higher than that of P2)

```
global s1 = new semaphore(1);
global s2 = new semaphore(1);

object HelloThread(1) {
 int threadID; this.threadID = 1;
 void run() {
  s1.P();
  Global.buffer2[Global.index2] = this.threadID;
  Global.index2++;
  s1.V();
  for (int i = 0; i < 3; i++) {
  s2.P();
    Global.buffer[Global.index] = this.threadID;
    Global.index++;
  s2.V();
  }} // run
} // end Thread(1)

object HelloThread(2) {
 int threadID; this.threadID = 2;
 void run() {
  for (int i = 0; i < 3; i++) {
  s2.P();
    Global.buffer[Global.index] = this.threadID;
    Global.index++;
  s2.V();
  }
  s1.P();
  Global.buffer2[Global.index2] = this.threadID;
  Global.index2++;
  s1.V();
  } // run
} // end Thread(2)
```

Figure 4.14: Two different sets of critical sections protected by two different semaphore objects, s1 and s2.

Bibliography

[1] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[2] Sreekaanth Isloor Anthony and T. Anthony Marsland. The deadlock problem: An overview. *IEEE Computer*, 13:58–78, 1980.

[3] M. Ben-Ari. *Principles of Concurrent Programming*. Prentice Hall Professional Technical Reference, 1982.

[4] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, June 1971.

[5] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, AFIPS '65 (Fall, part I), pages 185–196, New York, NY, USA, 1965. ACM.

[6] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with “readers” and “writers”. *Commun. ACM*, 14(10):667–668, October 1971.

[7] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, September 1965.

[8] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1(2):115–138, June 1971.

[9] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Comm. ACM*, 8(9):569, 1965.

[10] Edsger W. Dijkstra. A tutorial on the split binary semaphore. circulated privately, March 1979.

[11] Edsger W. Dijkstra. The superfluity of the general semaphore. circulated privately, April 1980.

[12] Edsger W. Dijkstra. Cooperating sequential processes. In Per Brinch Hansen, editor, *The Origin of Concurrent Programming*, pages 65–138. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[13] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996.

[14] Per Brinch Hansen, Edsger W. Dijkstra, and C. A. R. Hoare. *The Origins of Concurrent Programming: From Semaphores to Remote Procedure Calls*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[15] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, October 1974.

[16] Richard C. Holt. Some deadlock properties of computer systems. *ACM Comput. Surv.*, 4(3):179–196, September 1972.

[17] Scott Oaks and Henry Wong. *Java Threads, Third Edition*. O'Reilly Media, Inc., 3 edition, 2004.

[18] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[19] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

5
File Systems and I/Os

Computer users interact with a file system on a daily basis. Many everyday users would just consider computers as a file system in fact. You can click on a program icon to start it, or you can click on a document icon to open it with a word processor or a program that is associated with the document type. You make changes to a document, save it, and then expect it to be there tomorrow when you restart the computer. At times, you may want to recover files after accidentally deleted them. These actions are all possible because of a file system.

Application programmers also interact with a file system quite often. When a programmer wants to read from a file and write to a file, she uses I/O library calls to perform the desired operations. These I/O library calls invoke corresponding I/O system calls to delegate the work to the OS. There are two reasons for the delegation. One of them is to hide messy details of the lower-level I/O device operations from the application programmers. Application programmers should not have to know what kind of hard disk the computer has, for example, to read and write to and from the disk. Another reason is to protect the devices and OS from application programs by limiting access to the lower-level machine operations. Application programmers could otherwise damage or compromise the system either knowingly or unknowingly. The following are some I/O library functions in C that are used frequently. If you are a programmer, you have most likely used these calls.

- create(): create a file
- read(): read from a file
- write(): write to a file
- delete(): delete a file
- rename(): rename a file
- mkdir(): make a directory
- rmdir(): remove a directory

Independence of I/O library function call syntaxes from I/O hardware devices types

There are many different types of I/O devices, namely hard disk, keyboard, mouse, monitor, and network devices. Network devices are also I/O devices. Sending a packet is analogous to writing and receiving a packet is analogous to reading. I/O library functions and systems calls are written in such a way that even if the application programmers are performing I/Os for different types of devices, the syntax of I/O call should be the same. For example, the `read()` function has the syntax `read(fd, buffer, size)`. `fd` stands for the file descriptor which is a pointer to the file to read from. The data read will be stored in `buffer` in `size` amount. The same syntax can be used for reading (or receiving) from network. The only difference would be that `fd` now is pointing to a network stream (called network socket), rather than a file in a hard disk drive.

This design is intentional. There are many different I/O devices and it is impossible to come up with different I/O library syntax for different devices. In fact, it is even silly to do that. I/O devices fall into three different types: read only devices (keyboards, mouse, etc.), write only devices (monitor, cd rom, etc.), and read/write devices (hard disk drive, network, etc.) Therefore, we can unify the syntax of I/O library functions under these three types. Of course actual implementation of the system calls associated with different I/O library functions are different underneath OS.

Figure 5.1 shows an illustration of how a file system call from an application program is translated to the lower-level hard disk operations. After receiving a file system call, OS calls the subroutine[1] in the OS that interacts with the hard disk controller. As we mentioned in the I/O section (Section 1.6.7), the subroutine initializes the registers in the controller with proper values to control the hard disk hardware. If we are using the DMA, the subroutine will initialize the DMA controller's registers and the DMA controller

[1] the driver program

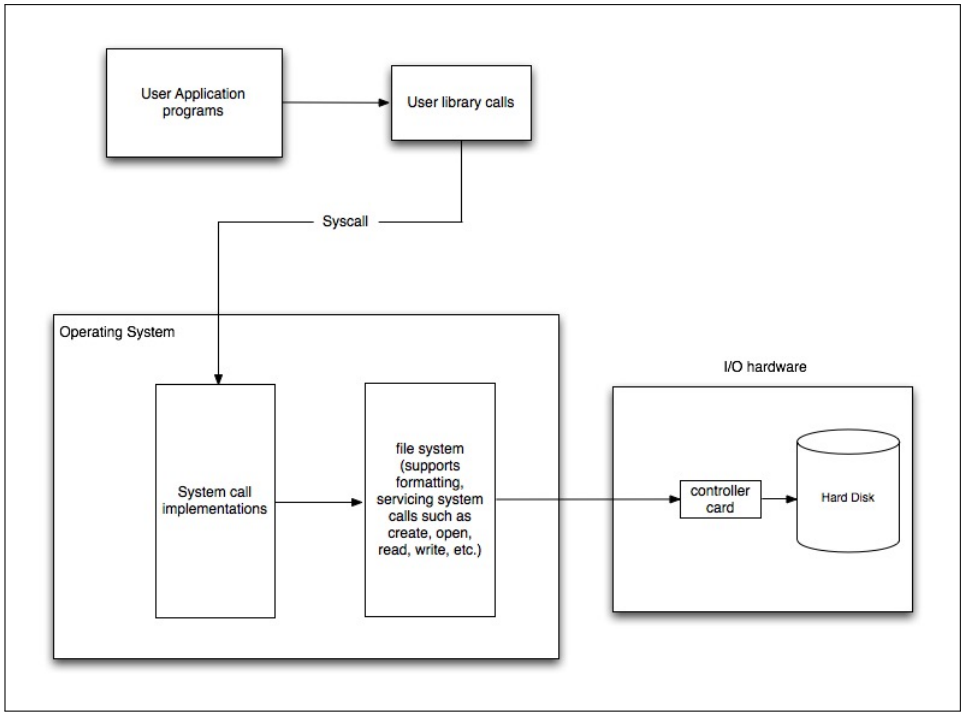


Figure 5.1: A flow of an I/O library function call from a user application program to an I/O device

will perform necessary I/O functions. Notice that the figure shows that I/O library functions are the interface between application programmers and OS.

The single most important thing that I want to remind you is to understand the various user perspectives of the file systems: the ordinary user, the application programmer, and the OS developers. OS developers design these system calls so that it is convenient for users and application programmers to perform various file related operations.

5.1 What is a File System For?

One of the most important reasons for having a file system is so that users can turn off the computers and expect to access files when they turn the computers back on. Unlike the main memory and cache memory, data in hard disks or solid state disks are non-volatile. All programs and data are kept in these storage devices without continuous power supplies..

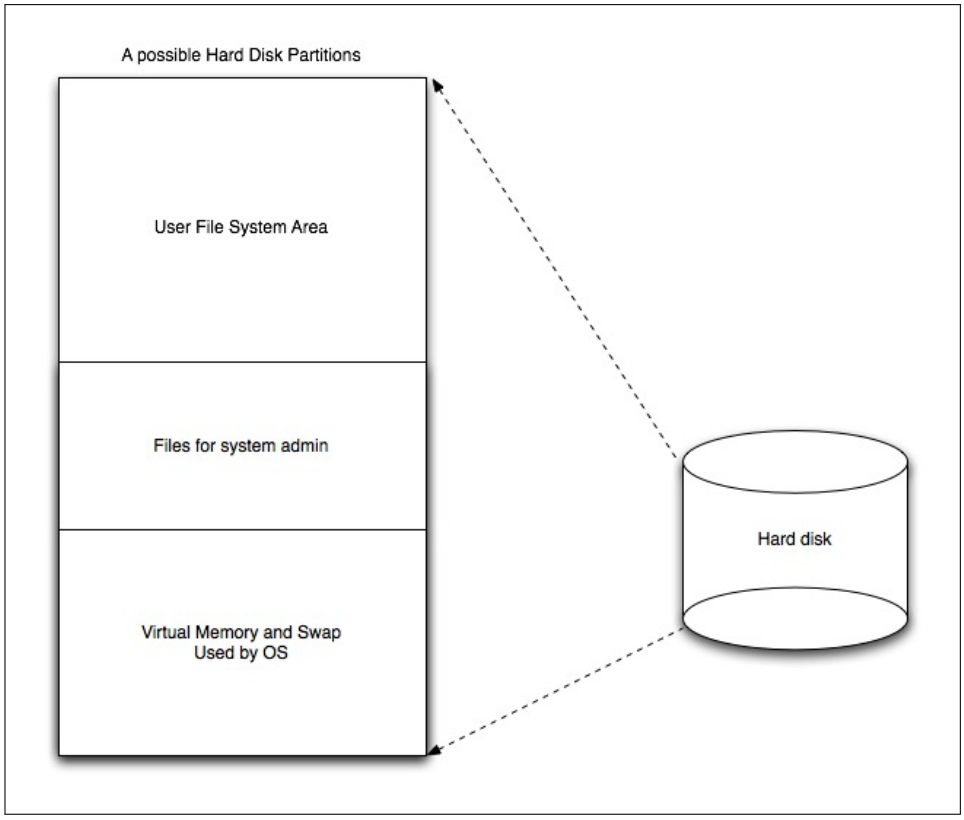


Figure 5.2: An example of hard disk partitions

5.1.1 Goals of a File System

There are several goals to consider in designing a file system:

- Store data in a non-volatile way
- Access data quickly and in various ways
- Save the storage space as much as possible
- Be able to recover from errors and crashes

The first goal is achieved through the hardware characteristics of the hard disk. A hard disk is made of magnetic discs that can retain data without electrical supplies. The second and third are, in some ways, conflicting goals. To understand this, we need to understand that there are roughly two types of information stored in the file system: meta-data and data.

Meta-data are needed to manage the file system. Meta-data provides information on how files are stored, the names of files, how to find files, etc. Meta-data are needed to access actual data. For example, imagine you are sorting and storing articles about computer science in a big steel file cabinet. How you arrange these articles in the file cabinet is up to you, but the goal is to design an indexing system so that you can find articles fast and reliably. The size of the file cabinet is limited, therefore you want to be able to store as many articles as you want. Notice that indexing information is meta-data. The indexing information should also be written and stored in the file cabinet so that anyone can find articles after reading how indexing has been done. One way to store articles in a file cabinet is to sort them in the alphabetical order in terms of the first authors. The index tabs may show the first letter of the articles. The index tabs are meta data. Actual articles are regular data. This indexing method works relatively well when the articles are only accessed through their author names. Now let's say your boss asks you to come up with an indexing system so that not only you can find a paper in the alphabetical order of author's name, but also in terms of the year of publication. You will then have to come up with a more elaborate method for indexing. The more elaborate the indexing is, the more complicated the indexing becomes; and consequently more book-keeping needs to be done. That is to say, more space is needed for indexing itself. Now the amount of meta-data would increase. If there is too much meta-data, then the file cabinet may not be able to store as many articles as it could. The same analogy works in the hard disk case. Therefore, we need to design with an efficient and effective way of indexing and managing the data using a minimal amount of meta-data.

5.1.2 The Memory Hierarchy and File System Design

In order to understand decisions regarding file system design, we need to review the concept of the memory hierarchy. We discussed the concept of memory hierarchy in the memory management chapter as well. As in Figure 5.3, the memory devices that are the smallest in their capacities are registers. Registers are in CPUs (and in device controllers). A CPU generally contains general purpose registers and some special purpose registers, such as the program counter. The general purpose registers are used to store temporary results of computations. For example, `ADD R0, R1, R2` instruction performs `R2 = R0+R1`. Many CPUs have 32 general purpose registers. Each register is usually one word. One word is usually 4 bytes. Next in the the memory hierarchy is L1 cache memory. It has a larger capacity than

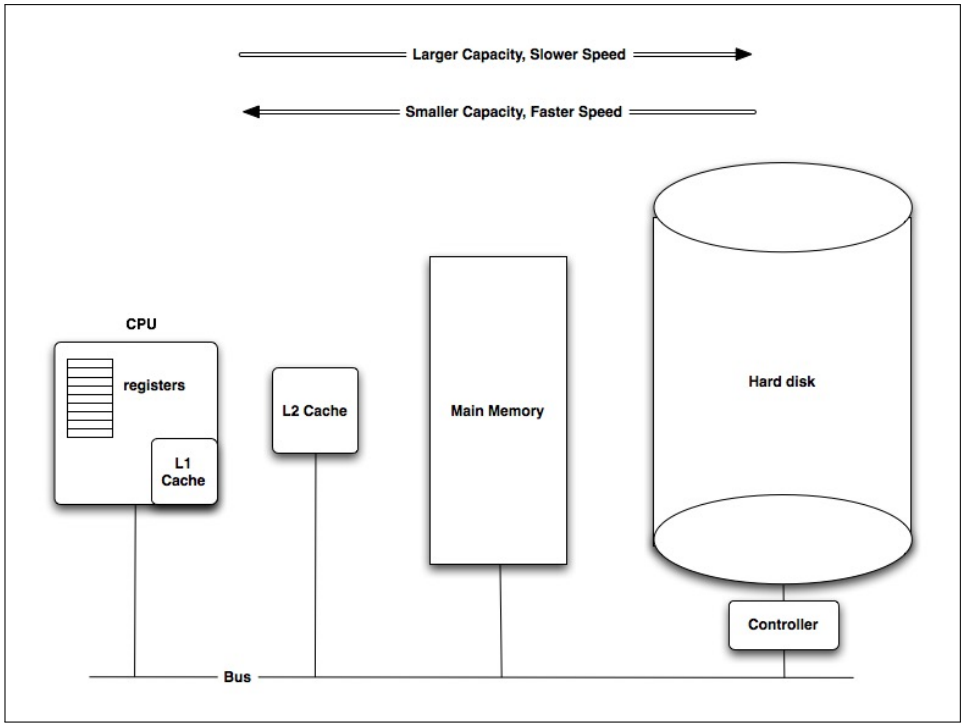


Figure 5.3: The memory hierarchy

the CPU registers combined. Typically Intel CPUs have about 32KB for data and instruction cache. The L1 cache is usually housed within the same package as the CPU. It is closer to the ALU and other CPU logic so that access time can be fast. The next in the hierarchy is the L2 cache. An L2 cache size can be 2MB but it is slower to access than L1. The speed of a memory device generally depends on two factors: it depends on the device's hardware characteristics, such as refresh rates. The speed is also directly related to the size. The hardware characteristics can be improved though better design and materials. However, size is a different matter. When the size grows, the control logic circuit size grows to cover the large number of memory elements in the memory device. Also accessing one particular element out of a large number of elements takes longer. Just think about finding a specific information on one page of paper compared to a large volume of papers. After L2, we have the main memory. Main memory can be easily several GB or even TB. As we just discussed, the access time of main memory would be even slower by just considering its size. Memory

access time is within the tens of nano seconds range. Traditionally, the next in the hierarchy are hard disk drives. Hard disk drives are huge and their sizes range from hundreds of GB to even Petabytes. The access speed of hard disk drives are extremely slow compared to the devices we discussed so far. It was over 10 milliseconds on average when this book was written. Notice that all other devices except the hard disks have their access time in nanosecond ranges. A millisecond is a million times slower than a nanosecond. The main reason for hard disks being so slow is because they are electro-mechanical devices. The mechanical part of hard disks slows down the device access time greatly. More recently, many computers come with Solid State Drives (SSD) instead of hard drives. SSD are faster than HDD, but still they are more expensive. Therefore, the capacity of SSD tends to be smaller in computers.

Being mindful about the characteristics of the various memory devices will help us to design file systems. As we discussed in the memory management, we have to avoid accessing hard disks as much as possible. We can do that by taking advantage of the principle of locality of references, as discussed in Chapter 3. However, avoiding hard disk access entirely is impossible. Why? When you turn off the computer, only the HDDs retain all your programs and data. Also, you have much larger amounts of programs and data than the main memory can store. Therefore, a file system must provide innovative ways to reduce data access time so the speed of the computer won't be slowed down by the slowest device in the memory hierarchy.

5.2 File System Design

In order to design an efficient file system, we must understand the hardware characteristics of the storage device. We will consider two types of storage devices in this book: Hard Disk Drive (HDD) and Solid State Drive (SSD).

5.2.1 Hard Disk

Figure 5.4 shows a disc of HDD. There are several discs in a HDD. A HDD is a sequence of disk tracks and sectors. In fact, the illustration is an abstraction of a real physical hard disk layout, showing a high-level logical view of the disc layout. For example, notice that a sector in an inside track (say, Sector 14, Track 9) is smaller than a sector belonging to an outside track (say, Sector 0, Track 0) in terms of area. Therefore, bigger sectors are physically divided into several sectors at the lowest-level of disk format. OS

designers rarely need to be concerned about this much of low-level abstraction. The control logic inside of the HDD package usually takes care of this and the HDD controller knows how to deal with such a low-level business.

Notice also that the HDD discs constantly spin when the HDD is turned on. There is a disk arm that moves from tracks to tracks. In order to access a specific sector, HDD must move the arm onto the right track (seek time delay), wait until the right sector comes underneath the arm (rotational delay), and access the sector to transfer the data to the disk controller and eventually to the main memory (transfer delay). There is a lot of nitty-gritty art of disk access in this low level, but we will leave it up for class discussions, if needed. One thing we must remember though, is the seek time is quite a delay due to the mechanical nature of the operation. There are many techniques for shortening the seek time delay. We will discuss how to alleviate the delays in data access caused by these three delay factors later.

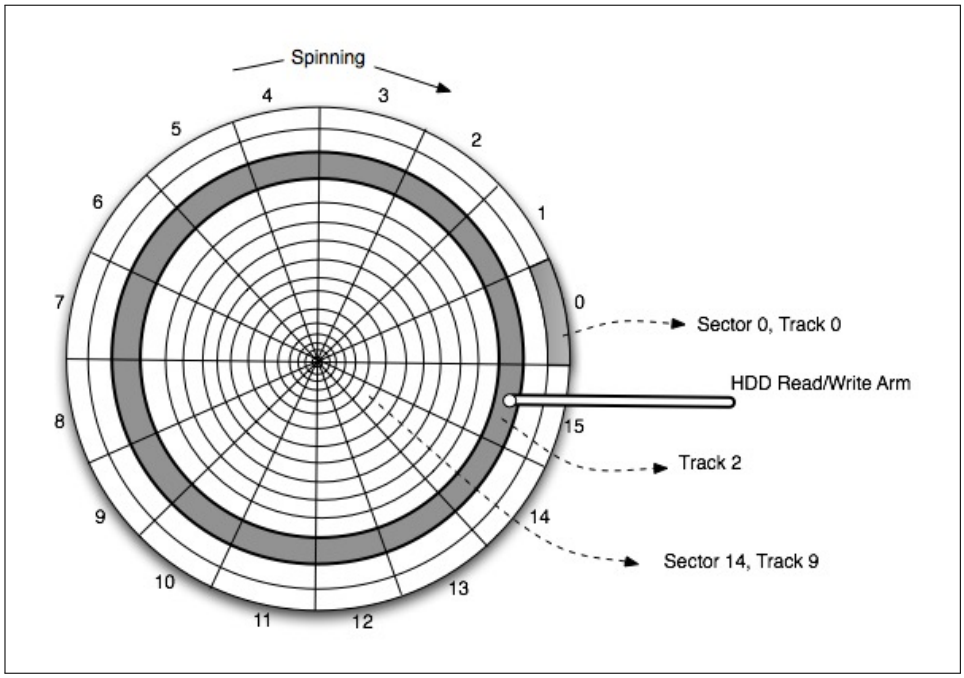


Figure 5.4: A disc layout of a typical Hard disk. Tracks are the circles and sectors are the divides within each track. There are 16 tracks and each track has 16 sectors in the illustration. This is of course a very small hard disk. Real hard disks have many more tracks and sectors.

An OS developer can see the hard disk as a linear sequence of disk sectors as shown in Figure 5.5. Furthermore, when reading from and writing to a hard disk, the file system often considers the hard disk as a sequence of *disk blocks.* A disk block is a logical unit for read and write from the OS's perspective. The size of a disk block is determined by the OS at the boot time. Often a sector can correspond to a disk block. However, a block can be larger than a sector for performance reasons. There are several reasons why an OS manages blocks rather than sectors. After all, using blocks means another mapping from block number to the right sectors. The block size is closely related to the performance issues in terms of time and space utilizations. A block is the smallest logical unit that the OS uses to access hard disk, which means if the block size is 2KB, then an OS will read and write for 2KB at a time, even if the OS needs only 2 bytes within that 2KB. So the larger the block size, the more data will be read or written in one access. Then why not make the block size really big? Then we will be reading too much unnecessary data for one access. If we make the block size too small, we will need more amount of meta-data. This is because we have to keep track of more blocks. There is an optimal block size one must compute in OS design. The "optimal size" can be computed using some mathematical formula, or decided by experiments using some benchmark programs. Because there are so many factors that contribute to hard disk access performance, finding the optimal size is a tricky business.

Make sure you understand the difference between a *sector* and a *block*.

Assume we have the 2KB block size for the HDD in the figure and the sector size is the same as the block size, the hard disk has capacity of 2KB $\times$ 16 tracks $\times$ 16 sectors, totaling 1,048,576 bytes = 1MB.[2] Unless we mention otherwise, we assume the sector size is the same as the block size in this book.

Consider sectors as page frames and blocks as pages. Sectors are the physical locations within a hard disk, while blocks are logical units that can be stored in a sector or more, much the same way as page frames are physical locations in the main memory, while pages are logical data units that can be stored in a page frame.

[2]Traditionally computer scientists use 1KB to represent 1,024 bytes, which is 2^10 bytes. However, except for this case, all other scientific notations use the "K" as $\times 1,000$. The International Electrotechnical Commission in 1998 came up with a new unit "Kibibyte" to denote the 1,024 bytes. We, however, will continue to use commonly used unit "KB" to represent 1,024 bytes.

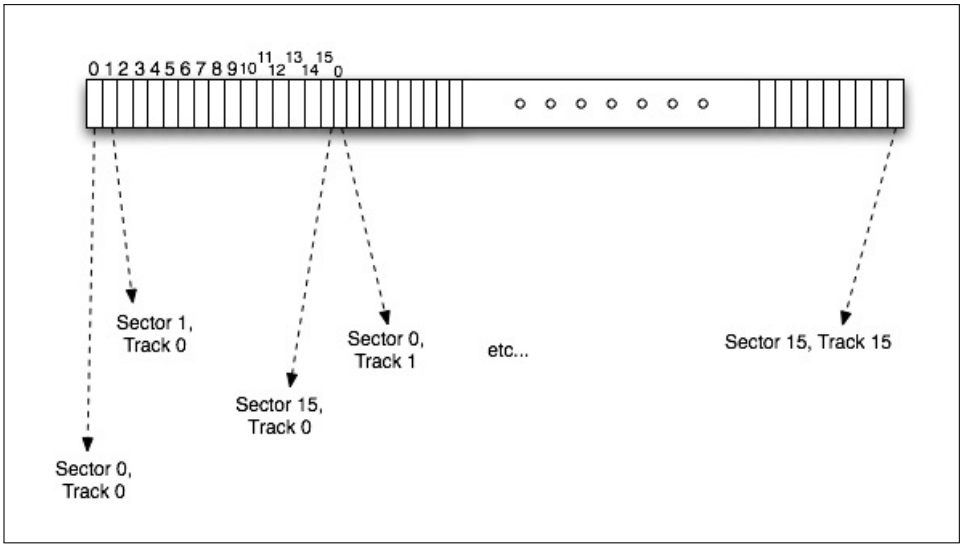


Figure 5.5: The OS designer's view of HDD layout. Showing sectors.

Character devices and Block devices

Roughly there are two types of I/O devices in terms of the access unit size. For example, keyboard and mouse are character devices while hard disk is a block device. Inputs from the keyboard are transferred one byte each time as a stream while inputs from (or output to) hard disk are one block at a time. Modems or network devices are character devices as well. Depending on whether the device is a character or block device, the way we design I/O driver routine can be different.

Hard Disk Block Size vs. Page Size

Some aspects of hard disk management are closely related to virtual memory management. For example, often the block size is closely related to the page size. For example, when a page fault occurs, the page needs to be brought in from the hard disk. If we make that reading one block would bring one page to the main memory, page fault service can be efficient. We can also arrange related pages closer to each other in the hard disk to reduce disk access time.

5.2.2 Storing Files in Hard Disk

When we store a file in a hard disk, we expect that file will be accessible next time we need it. Imagine one very simple way of storing files in hard disk. In this method, we start storing a file where the next available block is and we continuously save the file. For example, Figure 5.6 shows a method that stores a file in consecutive sectors, as many sectors as needed. The last block of a file is used to represent "end of file (eof)." For example, in the figure, if we assume one block is 2KB large, and file1 is 11KB large, we need 6 disk sectors, assuming the sector size is the same as the block size. Of course the sixth sector is only half used. The other 1K is an internal fragmentation within the last sector. This is a similar situation as in paging. In the figure, file2 requires 3 blocks, so three consecutive sectors are allocated.

Let's discuss pros and cons of this method. The obvious advantage of this method is simplicity. It is very simple to implement for an OS implementor. It also does not require much of meta-data. The only meta data needed are "eof" and a pointer (a variable) to the next block available. Let's say file1 belongs to the user John. John wants to re-access his file. How would OS assist John? One way would be to display the beginning part of each file in sequence asking John, one by one, whether it is the one he wants to access. This clumsiness is because we don't have file names to refer to files! One may say this method is useless. But it would be actually useful for extremely small devices without much space, especially when the device belongs to only one person and the number of files is small. Another issue with this consecutive allocation method is when deletion and insertion of files happens. Consider file2 being deleted in the example. Four sectors are free between file1 and file3. How do we reuse these sectors? It is clear that we can only store a file which needs less than 4 blocks in this "hole." Repeated creation and deletion of files will leave a lot of fragmentation like this in the hard disk. i.e., leaves many small consecutive sectors that are too small to accommodate any files. To resolve this issue, disk defragmentation process may be needed. However, disk defragmentation takes a long time. Requiring users to wait for a long time to create a file would frustrate the users. Disk defragmentation is useful to improve disk access performance when it is done during the night or when no users are using the computer.

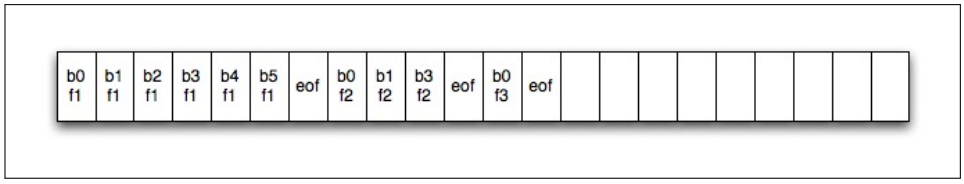


Figure 5.6: Contiguous file allocation with no support of using file names

How do we improve this file system so that it can refer to files with file names? Can we allocate disk blocks in non-consecutive sectors? We will discuss ways of doing this in .

5.2.3 Representing Directories to Enable File Names

Directory information is meta-data that contains information about files such as file names, sizes, where and how files can be accessed, etc. As we all know as computer users, if we double-click a folder, the folder is opened with a list of files in it. If it is a command line interface we are using, for example in Unix, we can use the command `ls` to list all the files in the directory. This command is possible because meta-data about the directory of files are kept. The table of contents at the beginning of this book can be viewed as directory information.

The directory information can be kept as a file. We call it a directory file. It is not easy to think about an alternative way of representing directory information rather than as a file. If someone asks you to come up with a data structure to manage directory information, what would that be? It would be most likely a table that contains the list of names of the files in the directory and information about the files. Each entry in the table would have *file name, file attributes, file size, first block address. File attributes* are information, such as access rights (for example, read only, read and write, executable, etc.). *First block address* is the first data block address of the file. This information would contain the sector and track number of the block.

```
class DirectoryEntry {
 char file_name[MaxFileNameLength+1];
   //+1 is for the terminating character, '\0'
 boolean read_permission;
 boolean write_permission;
 boolean execute_permssion;
 int filesize;
 int sector; // sector number for the first block
};
DirectoryEntry directory =
            new DirectoryEntry[MaxNumeFiles];
```

Figure 5.7: An example of directory entry definition and creating a directory object

Figure 5.7 shows this idea[3] Note that if we use the contiguous file allocation discussed above, we only need to find the first data block of a file to access all the data blocks belonging to it. Representing directory with a fixed size table such as this is simple. However, the table has a fixed number of entries and the file name length is also fixed. We can change the definition of `char file_name[MaxFileNameLength];` to `char* file_name` and use dynamic allocation of the `file_name` for variable length of file name. The number of entries in a directory can be also created and removed as needed so that there would be no limit to the number of files within a directory as far as the disk space allows it.

Now assume there is only one directory—the root directory—in this example. When the OS starts, it will read the directory file from a predetermined sector from the hard disk. Remember that all the file system structures, meta-data and data, must be stored in the hard disk. When the computer is turned back on, some of the meta-data are read into the main memory because they are accessed more frequently. The directory file is such information.

Representing File System: First Attempt

Let's see how the directory file is stored in the disk using the example in Figure 5.7. To see this, we first need to calculate the size of a directory

[3]This definition and some other definitions in this chapter are inspired by the file system code in Nachos.

entry. For the sake of an example, assume that `MaxFileNameLength = 7` and `MaxNumeFiles = 100`. That is, each file name must be 7 characters long or less. This means that `file_name[MaxFileNameLength+1]` is 8 characters total, including the end of string character `'\0'`. Each character is usually 1 byte. Therefore, we need 8 bytes for a file name. There are three boolean variables for permissions. Each boolean variable is 1 byte in C, totaling 3 bytes, `int filesize` is 4 bytes in C as well as the size of `int sector`. Therefore, the total size of a directory entry is $8 + 3 + 4 + 4 = 19$ bytes. Since 19 is not a multiple of power of 2, the next smallest number in the power of 2 series is 32. Therefore, we need 32 bytes for one directory entry. The 13 bytes out of that 32 bytes are not used. We can go back to the design of the directory entry and reduce the entry size to 16 by sacrificing some information in the entry or we can introduce more attributes to utilize the 13 bytes. However, for now, as an example, we will keep the current definition of directory entry. Figure 5.8 shows this design.

Since one entry needs 32 bytes and we wanted the maximum number of files in our file system to be 100 files, the total number of bytes for the directory table is 3,200 bytes. Now we need to store this 3,200 bytes in the hard disk. Our hard disk has block size 2KB = 2,048 bytes. This means that we need 2 blocks = 4,096 bytes to store the directory file. Again $4,096 - 3,200 = 896$ bytes that are not utilized in the second block.

During a disk formatting process, the file system subroutine in OS will write this directory table to the hard disk. Since we are assuming the contiguous file allocation scheme discussed above, this structure would be all we need for meta-data. In order to find a file with its name, say `project1.c` for example, the file system will search the directory table entries to find an entry with the same file name, i.e., `project1.c`, then it will read the value of the `sector` field in that entry to find the first block of the file. Because the files are stored in the hard disk in the consecutive disk sectors, finding the first block is enough to access the entire body of the file.

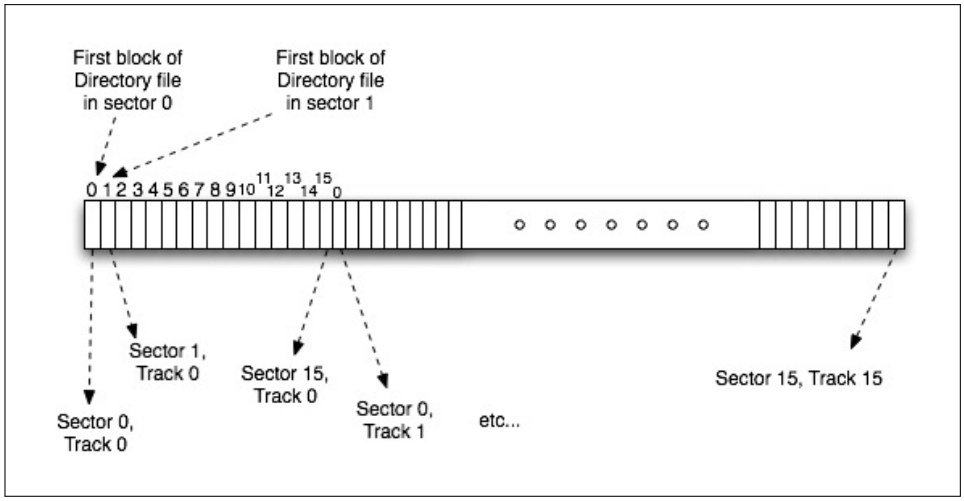


Figure 5.8: A simple directory data structure stored in our HDD example. The OS always knows where to find the directory structure file (sector 0) and the how large the file is (2 blocks)

Representing File System: Second Attempt

One problem we discussed about having a contiguous block allocation is the problem of fragmentation. The defragmentation improves disk speed but it has a heavy overhead. Let's see how we can accomplish *non-contiguous allocation,* so that file blocks don't have to be allocated in a sequence of consecutive sectors.

The first method is used in old DOS system using the FAT (File Allocation Table). We will still use the same directory structure for this example. Figure 5.9 shows this idea. FAT is stored in predetermined sectors of HDD. We need 256 entries in the FAT since 16 sectors/track $\times$ 16 tracks is 256 sectors. Each entry in FAT is an integer, which is 4 bytes. 4 bytes $\times$ 256 blocks = 1,024 bytes = 1KB. Since 1 block (or sector) is 2KB, we would need 1 disk sector for storing the FAT, which is stored in block 2 in the figure. Block 0 and 1 are used for the directory file as in the previous example. The FAT allows non-contiguous file allocation. For example, file `proj1.c` in the figure starts from sector 5. The FAT's entry 5 has 3, therefore following the entry 3 in FAT, we find 6 in, and finally following 6, we find an `eof`.. Therefore, we know that file `proj1.c` is stored in disk sectors 5, 3, and 6, in that sequence, total 3 blocks large.

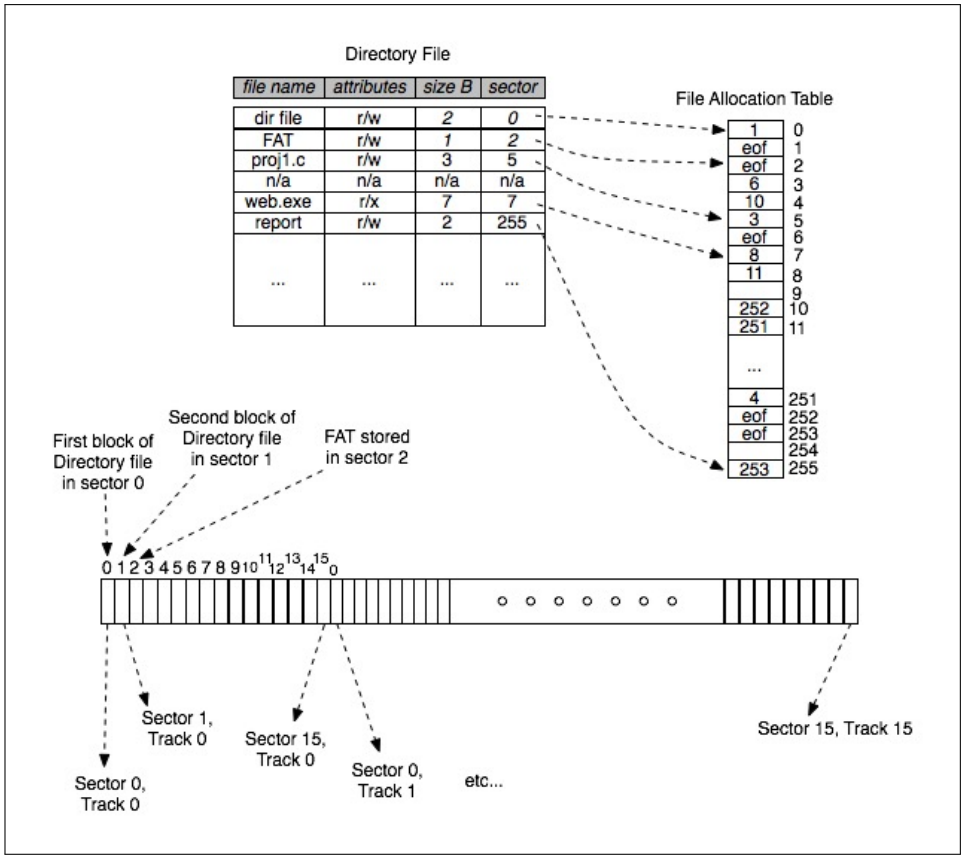


Figure 5.9: An example of File Allocation Table.

Representing File System: Third Attempt

One of the disadvantages of FAT is when the sector storing the FAT is lost, all the information about all files are lost. In our example, if the disk crashes and somehow sector 2 is damaged, FAT is lost forever. We could replicate the FAT in another sector, but we also can come up with a better method.

Unix and its variants use the notion of i-nodes. I-nodes are also known as file headers. Each file is represented by an i-node. An i-node contains information about the file it represents. Theoretically, we can move a lot of information about files from the directory file to i-node. The directory file could just have the file name and the sector number of the i-node. That way, even if the directory file is damaged, i-nodes still contain a lot of information about files they represent. Of course, if the directory file is completely

destroyed, the OS somehow needs to find i-nodes by scanning the hard disk. Since the OS knows what i-node data structure looks like, it can find i-nodes scattered in a HDD relatively easily. Figure 5.10 shows the new directory entry. Now each entry is 8 bytes + 4 bytes = 12 bytes. 12 bytes × 100 entries = 1,200 bytes, which requires only one disk block (2K) as opposed to two disk blocks in our old definition of directory entry. The directory entry is simplified because now i-nodes can contain information about files. Each directory entry mainly enables the OS to find the corresponding i-node.

```
class DirectoryEntry {
 char file_name[MaxFileNameLength+1];
   //+1 is for the terminating character, '\0'
 int sector; // sector number for the first block
};
DirectoryEntry directory =
            new DirectoryEntry[MaxNumeFiles];
```

Figure 5.10: A simplified directory entry for i-node based file systems

Figure 5.12 shows an example of i-node (a.k.a. file header). It is usually a good idea to make the i-node to fit into one disk block, i.e., 2KB. There are $\lfloor(2048-(3+4+4))/4\rfloor = 509$ direct block pointers. This value is computed in the following way: One block is 2KB, so it is 2,048 bytes. There are two integer variables that require 8 bytes total. The three boolean variables sum up to 3 bytes. Since each pointer to data blocks (represented by `int directBlocks`) must be 4 bytes, we need to divide the resulting value by 4 and take the floor function of the result, which yields to 509 pointers. Since there are 509 pointers that can point to data blocks, this i-node definition can create a file size as big as 509 × 2KB = 1,018KB. Figure 5.11 For this disk of size 512KB, this is more than enough and we can live with this i-node design.

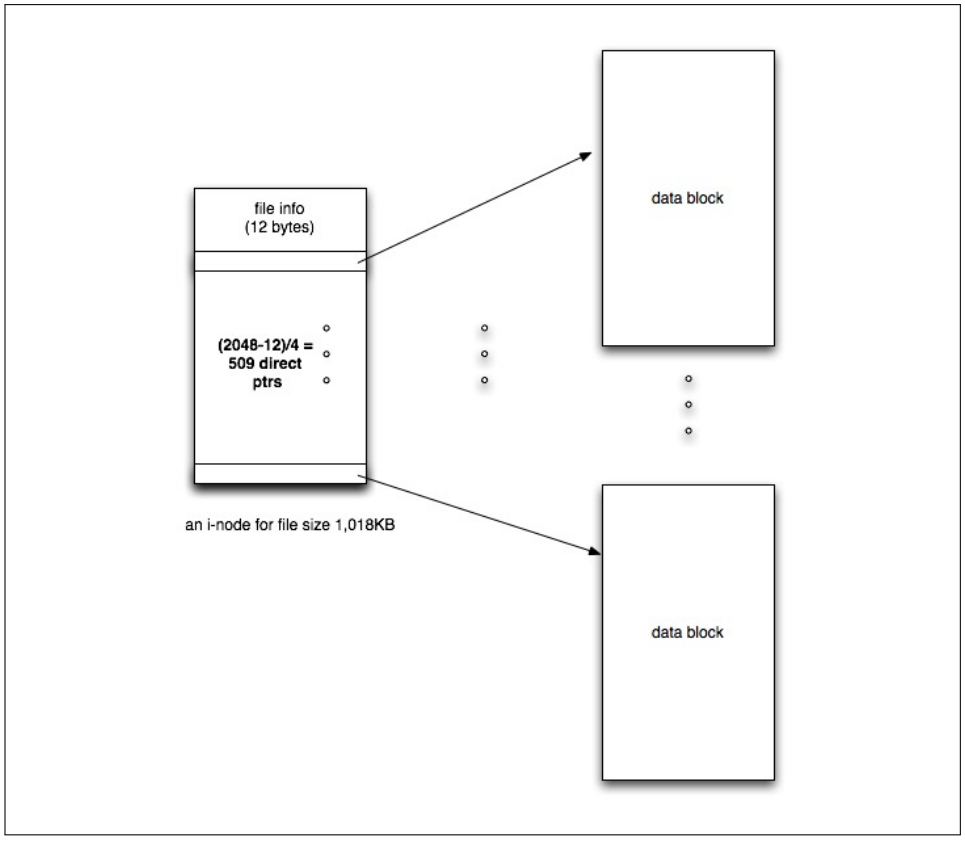


Figure 5.11: An i-node that only uses direct block pointers. There are 509 direct block pointers. Therefore, using this i-node format, we can create a file as big as 509 $\times$ 2KB = 1,018KB.

```
class i-node {
boolean read_permission;
boolean write_permission;
boolean execute_permssion;
int fileSizeInBytes;
int fileSizeInBlocks;
int directBlocks[floor((2048 - 3 - 4 - 4)/4)];
}
```

Figure 5.12: An example of i-node definition

What if, however, if we went out and bought a HDD with a larger capacity, say 16,384KB = 128MB? We want to create files larger than 1,018KB as far as there's enough disk space. Some may suggest to use multiple i-nodes, one for the first 509 data blocks, the second i-node to represent another 509 blocks, etc. If we use this method, we are unnecessarily replicating other meta-data such as permissions booleans and file size in bytes and blocks. Once we have one i-node, we can expand the pointer capacity of the i-node with indirect pointers as shown in Figure 5.13. The i-node definition is modified to have additional data blocks through indirect block pointers.

```
class i-node-improved {
boolean read_permission;
boolean write_permission;
boolean execute_permssion;
int fileSizeInBytes;
int fileSizeInBlocks;
int directBlocks[floor((2048 - 3 - 4 - 4)/4) - 2];
int singleIndirectBlocks;
int doubleIndirectBlocks;
}
```

Figure 5.13: An example of i-node definition using direct block pointers, an indirect block pointer, and a doubly indirect block pointer.

Let's see how big of a file size this i-node can create. There are a total of 507 direct block pointers. One single indirect block pointer will point to a block that is entirely data block pointers. How many pointers does this block have? That's 2KB / 4 = 512 pointers. Using a double indirect block pointer, we can have 512 $\times$ 512 = 262,144 pointers. Summing up all the available pointers, 507 + 512 + 262,144 = 263,163 pointers in total that an i-node can reference. If a disk block size is 2KB, the largest file size an i-node can manage is 263,163 $\times$ 2KB = 4.0155GB !!

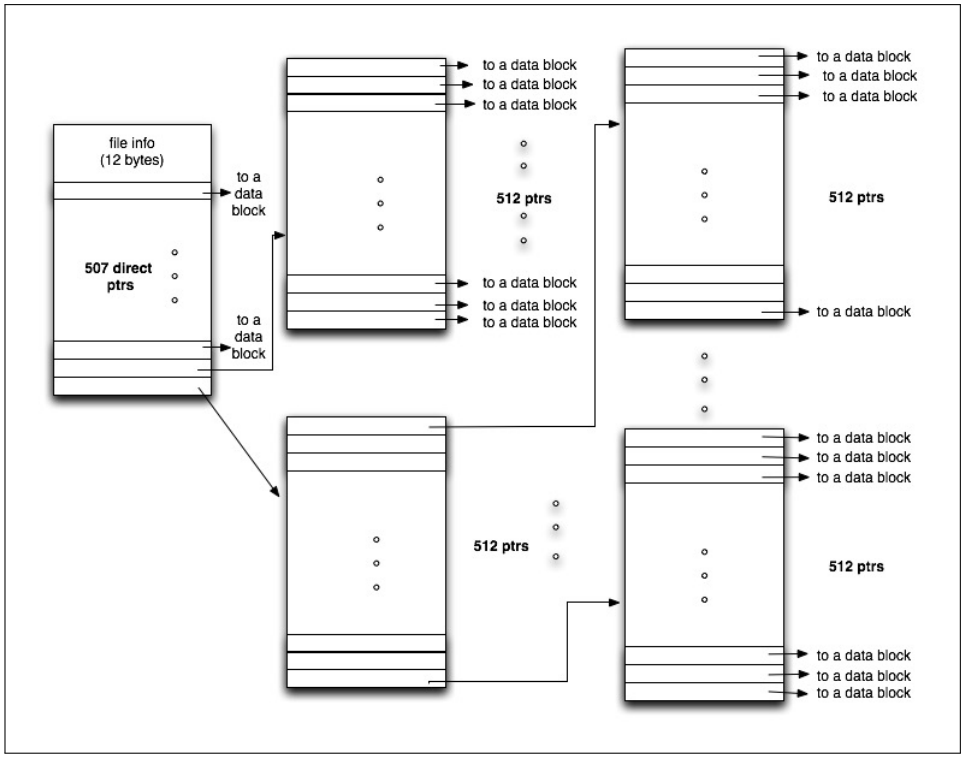


Figure 5.14: An illustration of how i-node can be used to manage huge files by utilizing direct, single indirect, and double indirect pointers.

Figure 5.14 shows the diagram of the above i-node configuration. Notice that the i-node itself can address up to 507 data blocks. If the file requires more than 507 $\times$ 2KB, we can use the single indirect block pointer that points to a block of 512 pointers to data blocks. We can create even larger files using the double indirect block pointer that can point to up to 512 blocks of pointers. Real OSs even use triple indirect block pointers!

Another advantage of the i-node method is that the OS does not have to allocate all data blocks from the beginning. If a file requires only one data block, it would just need its i-node and one data block. Later the same file can get larger; for example, the user added more data to the file. Then it will simply get another data block and that block is pointed by the second entry of the `directBlocks` array. Figure 5.15 shows the i-node file system of the situation in Figure 5.9. Note that the directory entry for FAT is not needed for the i-node example and some file data blocks are different in the

i-node example from the FAT example to accommodate file headers in the i-node scheme. FAT does not need file headers.

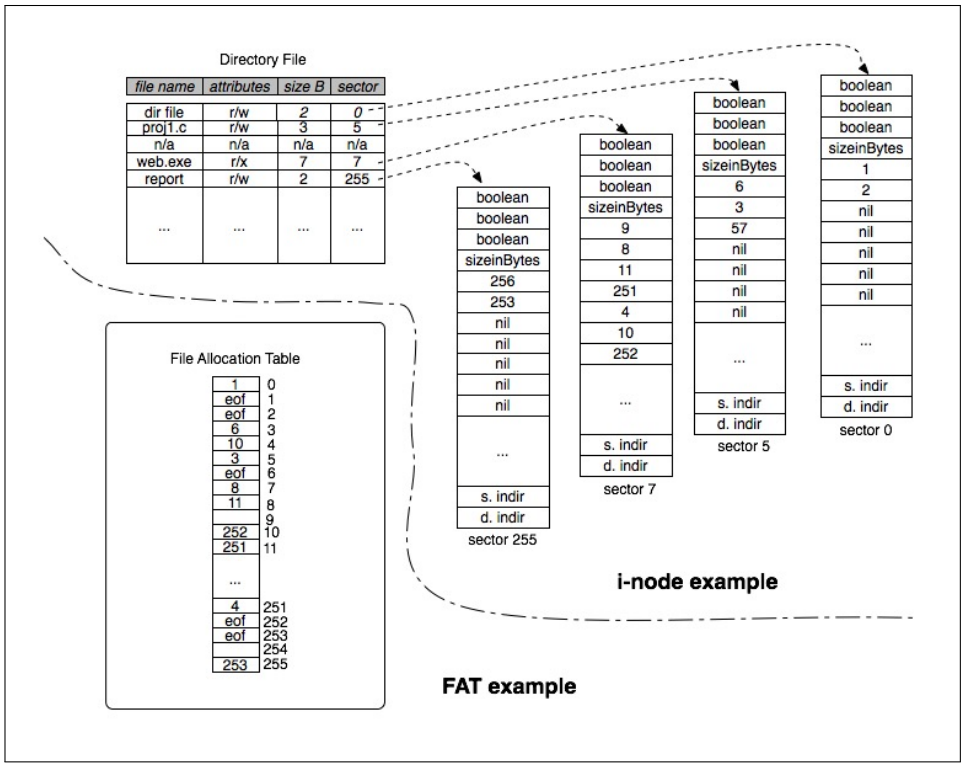


Figure 5.15: An equivalent i-node file system as the FAT example in Figure 5.9.

5.3 Keeping Track of Free and Used Sectors

There are several ways to keep track of which sectors are being used and which are free. We will discuss one method that uses a bitmap. A bitmap is a sequence of bits in which each bit represents whether the associated resource to that bit is available or in use. For example, we can use a bitmap to represent which sector in the hard disk is available or not. Let's design a bitmap for our hard disk of 16 tracks and 16 sectors per track. There are $16 \times 16 = 256$ sectors. Since one block is 2KB and one block is the smallest unit of access, one block is enough to host a bitmap of 256 bits. Since the sector size is the same as the block size in our example, one sector

will contain the bitmap. To use a consistent convention, we will represent the bitmap structure as a file, which means that it will have it's own file header (i-node) and the file body (file blocks). The bitmap file therefore will have its i-node (1 block) and its file body (1 block).

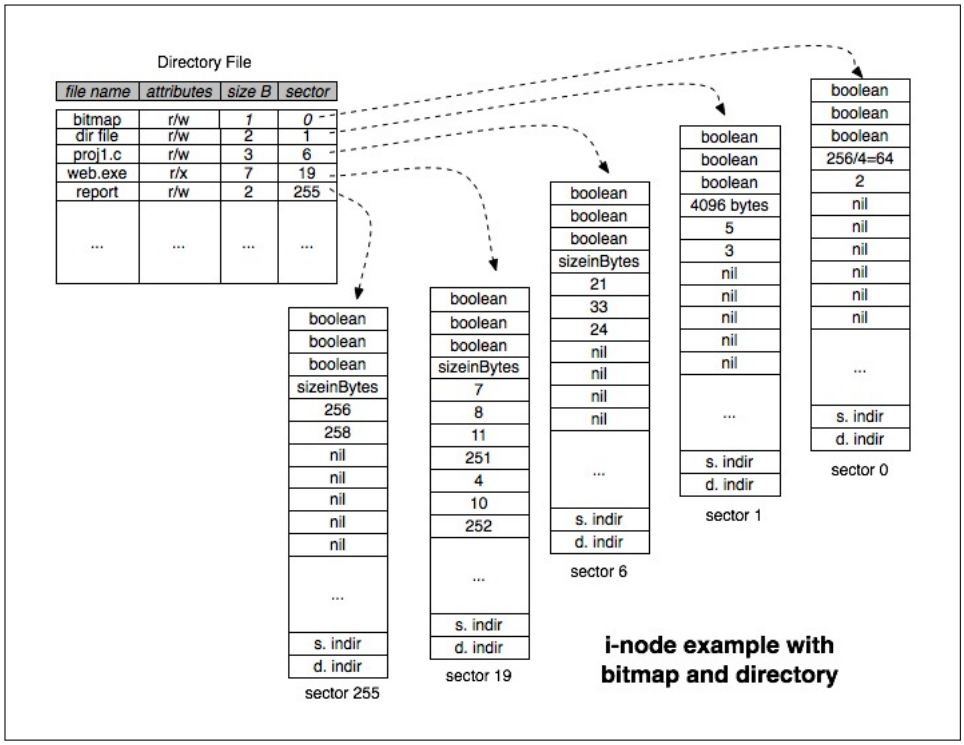


Figure 5.16: A complete disk map including bitmap, directory, and other files

We will also treat the directory table as a file with its file header and the body. As computed above, we need 1 block for the directory file header and 1 block for the directory file body. These two files, the directory file and the bitmap file are special files that are needed frequently by the OS when the computer is on. Therefore, they have to be loaded as the computer boots up to the main memory. The file headers for these two special files, therefore, should be stored in a predetermined location. We assume that we store the bitmap file header at sector 0 and the directory file header at sector 1. Then the bitmap file body is at sector 2 and the directory file body is at sector

3.[4] Figure 5.16 shows this example. Note that the file blocks for some of the files have been changed to accommodate the bitmap file header and the bitmap file block itself. The idea is that file blocks can be stored in any available sectors in the disk as the file system keeps track of the bitmap file.

5.4 Performance Issues of File Systems

Now that we know several methods of constructing a file system, we should turn our attention to improving the performance of file systems. In order to do this, we need to understand the disk data access patterns of typical application programs.

Remember two things as you read below when we discuss performance issues of file systems: (1) the principle of locality; and (2) accessing hard disk is 10^6 times slower than accessing memory.[5]

There are generally four ways for improving file system speed performance:

- Use a Disk Cache: Computer systems with a large amount of main memory can set aside a chunk of main memory to serve as disk cache. This is in fact injecting another device between main memory and hard disk in the memory hierarchy. The disk cache will be formatted as if it is a collection of disk sectors. Frequently used blocks are brought into the disk cache for fast access.

- Find the optimal disk block size: The smallest unit of access in HDDs is a disk block. A HDD reads and writes one block at a time. Given this, OS developers can find the optimal size of hard disk block so that I/O request performance can be improved. For example, when a page fault occurs, the faulted page must be brought in from the HDD. The disk block size should be designed carefully. If the block size is too big, unnecessary data can be accessed along with the relevant data. If the size is too small, multiple read may be needed, which will significantly cost the performance.

- Optimize seek, rotational, and transfer delays: HDD access time is computed by the sum of seek, rotational, and transfer delays.

[4] This simple example is somewhat inspired by the file system format in the educational operating system called Nachos developed by UC Berkeley.

[5] We should also remember that the closer the memory device gets to the CPU, the faster and smaller in capacity it gets.

- Seek delay is the time that it takes the head of the read/write arm of the hard disk to travel to the target track, on average. In early days of manufacturing HDDs, the seek delay could be as much as $600ms$. The technology has been improving and recently we see it ranges from $4ms$ to $15ms$ from high-end HDDs to lower-ends. By scheduling the arm movement and placing the blocks that are accessed more frequently to a certain area on the disc of HDD, we can minimize the average seek time over a large enough number of disk accesses. We will discuss methods for this below.
- Rotational delay is the average time it takes for the disc to rotate so that the sector to be accessed comes under the read/write head. The average rotational delay is computed as the time it takes the disc to rotate a half-revolution times the revolution per minute (rpm) of the disc. For example, if a disc rotates 1,000 rotations per second, it takes $6ms$ per rotation. Therefore, the average rotational delay is $3ms$.
- Transfer delay: Transfer delay is the time it takes data to be transferred from/to disk to/from the I/O memory buffer. Overall, the transfer rate is about 1,030 Mbits/second. This is approximately 128 MB/s. If we have 512K block size, approximately 250 blocks are accessed per second from HDD to memory, vice versa. How do we utilize the above characteristics of hard disk hardware to improve the HDD access time? Consider the I/O command `write(filedesc, buffer, amount)`: It's easy to see that we rarely write to HDD in the amount of 250 blocks per second (or 128MB/s) for one I/O command. This means that executing this command by itself will under-utilize the disk bandwidth. Therefore, OS should not try to access HDD for just one or even a small number of I/O request. It should wait until a high enough number of I/O requests are collected then flush all of the I/O requests collected to HDD at once. Therefore, many writes to HDDs are actually done in the disk cache. Occasionally the cache is flushed to the actual HDD. Study the `flush()` command in C. For read operations, if the desired block is found in disk cache, we don't need to access HDD. Since the size of disk cache is much smaller than HDD's, similar algorithms like page replacement algorithms are used to bring in new disk blocks to disk cache when there is no available disk cache block.

- New file system design: We can carefully study the technology trends,

such as new storage devices and memory devices, and design a new file system that can take advantage of the new hardware technology trends. As an example, we will discuss how we can improve file access using SSDs (Solid State Disks).

5.4.1 Optimizing the Seek Delay

In earlier days in HDD hardware development, the seek delay was the longest delay among the three HDD delays. To reduce the average distance for the disk arm to travel, we can use the following two methods.

Disk Arm Scheduling

Imagine the situation that there are I/O requests for disk tracks, 1,10, 4, 5, 11, 4, 9, 15, 0, and 7. How do we minimize the distance the arm has to travel? One algorithm is called the *elevator algorithm.* This algorithm will first sort the request to 0, 1, 4, 4, 5, 7, 9, 10, 11, and 15. Then sequentially serve the requests. Compare the distance the arm has to travel if it has to serve the original sequence. It is easy to see the original sequence yields longer distance to travel. What if we don't have all these request in the beginning? Can we run this algorithm as disk access requests come in? Yes we can. Assume we have the first request at track 5. The arm will move there first. As it serves the request at track 5, two other requests at track 2 and 4 come along in that order in time, so we move the arm to track 4 first, then move it to 2.

Another algorithm is called the *shortest seek time first (SSF).* The SSF algorithm will serve the next track that is closest to the current track being served. For example, if the arm head is on track 7, and two other requests at track 8 and 1 come along. It will serve track 8 first because it is the closest from 7 at the time. There are a variety of algorithms for disk arm scheduling.

Organ Pipe Distribution of Disk Blocks

Some disk blocks are more frequently accessed than others. For example, some meta-data blocks such as i-node, directory blocks, etc. are so called "hot blocks." These hot blocks are, on average, accessed more often than other blocks. There can also be other data blocks that are accessed more often. Consider a compute server computer that is used by many users. Some users login more often and use the computer more often and for a

longer period of time. Disk blocks belonging to these users would be accessed more often.

The organ pipe distribution algorithm (OPD) places hot blocks in the middle tracks so that average distance traveled by the arm is minimized.

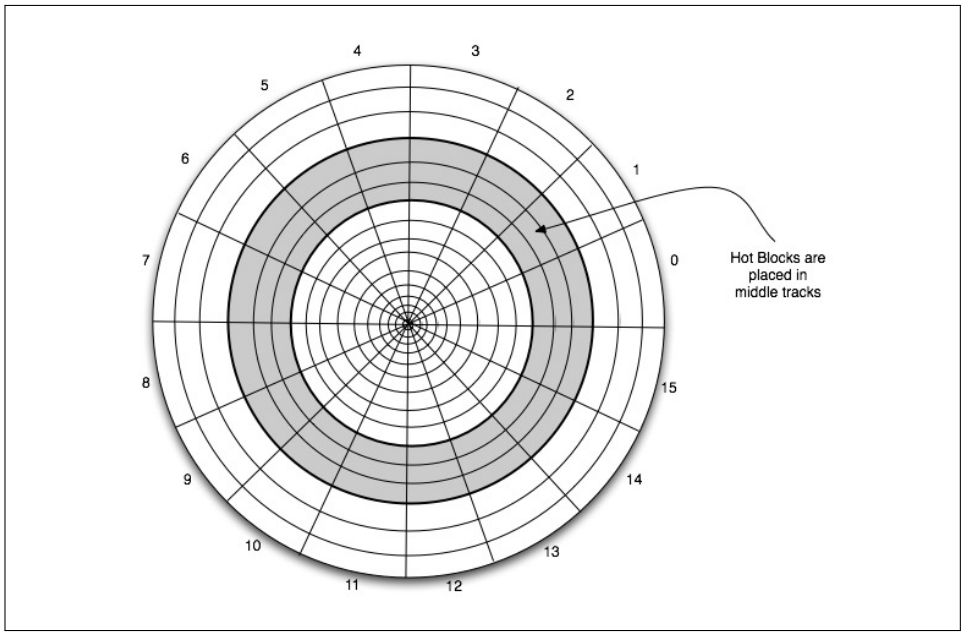


Figure 5.17: Organ Pipe Distribution Scheme to reduce disk access time. The average seek time is optimized

Figure 5.17 shows this concept. When the HDD is started, the arm head is moved to the center track. Because the blocks stored in the middle tracks are more frequently accessed, the disk arm would stay in middle track for the most of times, minimizing the average distance traveled.

How does OS identify hot blocks? One way is to keep a counter variable associated with each block in the HDD. Every time a block is accessed, the associated counter is incremented. In the middle of the night, when no users are logged in, the OS can rearrange disk blocks according to the counters.[6]

[6]In fact, some people say this method wouldn't work because many computer users (programmers) are nocturnal so that the middle of night would be the time that the computer is heavily used.

5.4.2 Optimizing Rotational Delay

The rotational delay is the time it takes for the desired sector coming under the read/write head. Because the disk rotates quite fast (1,000 rpm), logically sequential blocks could be stored in every other sector to improve disk access speed. This technique is called *sector interleaving.* Figure 5.18 shows this concept of two different inter-leaving techniques.

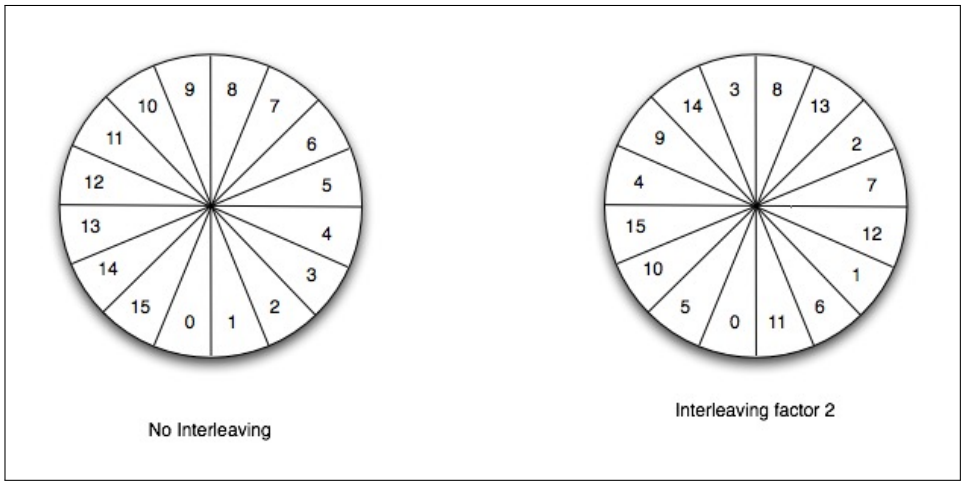


Figure 5.18: Using disk Interleaving to continuously access logically consecutive blocks without waiting for the next full rotation of disk. The interleaving factor (how many sectors to skip) is depending on the rotational speed.

5.4.3 New File System Design to Improve Disk Write Operations

Let's check the current technology trends in storage and memory devices. In terms of hard disk drives, the capacity of them has increased dramatically. Average consumers can buy 2TB capacity HDDs with several hundreds of dollars. However, the speed of access has not improved much recently. The main memory sizes have grown quite a bit for consumer grade computers ranging in several GBs. This means that OSs can utilize a larger disk cache. Statistical studies show that most of reads can be done by accessing a disk cache given the cache management is done properly. For writes, we need to eventually write to the actual HDD even if we write to disk cache first. In other words, we can write to disk cache until the substantial part of disk

cache is 'dirty.' Then we can write a big chuck of data from disk cache to the HDD at once.

SSDs became more common although the capacities are limited to several hundreds of GBs. More computers are using SSDs nowadays. SSDs lack mechanical parts so that it is much faster than traditional HDDs. SSDs have the characteristic that require to erase and reprogram a large amount of space before it can write. Each cell in an SSD has limited erase/program cycles. In SSDs, write operations are much more expensive than read operations. This is consistent with the current trend of HDDs using a large amount of disk cache. Since SSDs have no mechanical components, data locations wont make difference in performance. Therefore, optimizations on data locations are no longer helpful (think organ pipe optimization, disk arm movement optimization, etc.). Due to the write operation characteristics in SSD, random writes are extremely slow and bad for the life span of SSD.

In summary, for both SSD and the HDD, the current technology trends call for efficient write operations while read operations are done through disk cache most of times.

Log-structured file systems attempt to minimize overhead of write operations by buffering writes to disk cache; when writing to HDD is needed, it writes sequentially at the end of log. A small writes are buffered as a segment in disk cache then the entire segment is written to the HDD at the end of HHDs log. Because writes are done in contiguous disk blocks, seek time and rotational delays are reduced; disk bandwidth is also well-utilized.

Managing contiguous blocks and disk log requires some detailed techniques but the overall idea is simple.

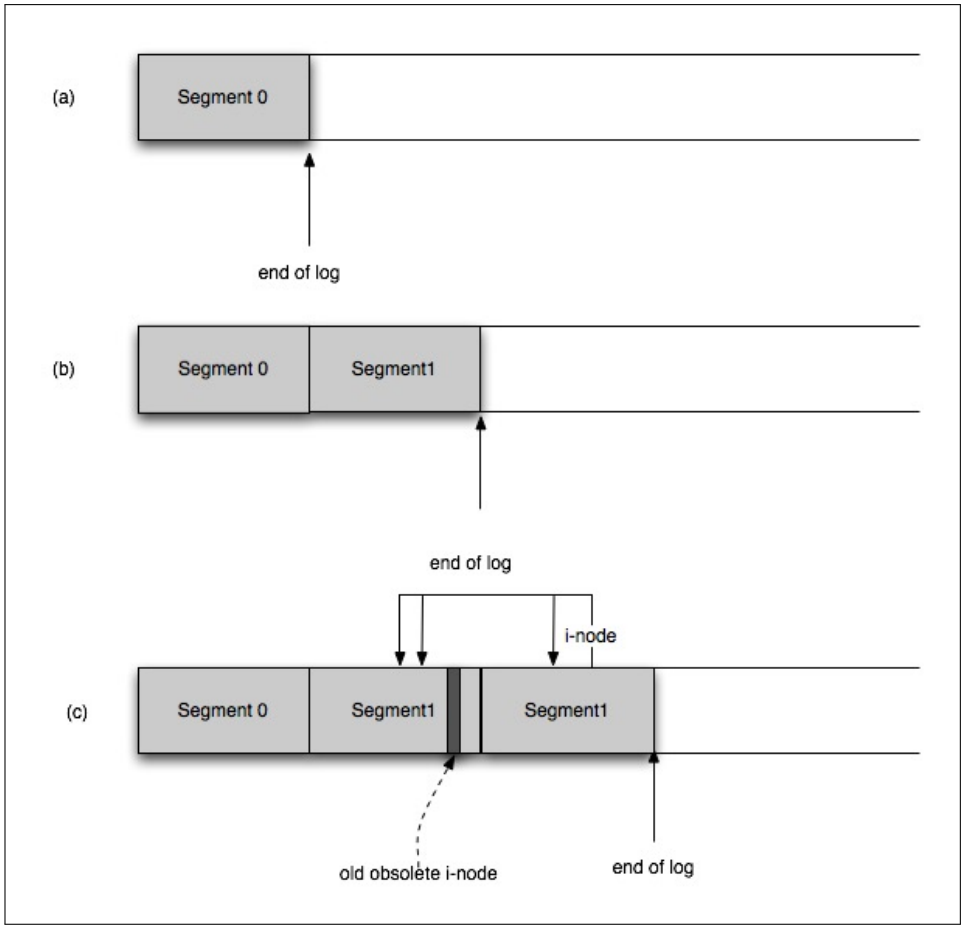


Figure 5.19: An example of Log-structured file system

Figure 5.19 shows an example of LFS operations. LFS writes to the disk one segment at a time. A segment is a collection of disk blocks and some meta-data to manage the segment, as well as the blocks belonging to it. At first, segments are written within disk cache, so there is no actual disk accesses. When a segment is full, the segment is written sequentially from the end of log (See (a) and (b)). In an ideal situation, segments are all "clean," meaning that there are no invalid blocks within segments. However, in practice, some blocks in segments can be irrelevant. For example, in Figure 5.19-(c), a new i-node is created for a file in segment 2 because a new block is added to the file. Because the two original blocks belonging to the file is still in the old segment, the new i-node points to the two

original blocks in segment 1 and the new block in segment 2. As these kinds of updates repeat, old segments become fragmented, meaning more portions of the segment become invalid. This means that read operations will make the disk arms to move back and forth among segments. The segment cleaner daemon process can run occasionally and combine most fragmented segments into one solid segment, writing it at the end of log, of course sequentially. When segments are less fragmented, read operations are more efficient because reading multiple blocks can be done reading one or more physically consecutive segments.

Exercises

Ex. 5.1 — Note the disk block allocation illustration in Figure 5.9 for FAT. Show a similar illustration for the i-node example in Figure 5.15.

Ex. 5.2 — Generally, I/O commands are not considered part of a programming language. Explain why this is so.

Ex. 5.3 — Discuss differences between blocking and non-blocking system calls.

Ex. 5.4 — Discuss one non-blocking system call and how it is useful for programmers.

Ex. 5.5 — What is the precise difference between I/O library calls and I/O system calls?

Ex. 5.6 — Consider `read(fd, buffer, size)` call in an interrupt-driven I/O but not using DMA. Explain, in as much detail as possible, step-by-step, what happens when the call is made by a process, say P0, until P0 runs again after the I/O call has been completed.

Ex. 5.7 — Consider `read(fd, buffer, size)` call in an interrupt-driven I/O and using DMA. Explain, in as much detail as possible, step-by-step, what happens when the call is made by a process, say P0, until P0 runs again after the I/O call has been completed.

Ex. 5.8 — Consider `read(fd, buffer, size)` call in an interrupt-driven I/O using DMA and without using DMA. What are the differences?

Ex. 5.9 — Consider Figure 5.1. In the box labeled "Operating System," if the system call implementation occasionally directly accesses I/O hardware bypassing "file system," what consequences do you expect?

Ex. 5.10 — Hard disk space contains the file system. There is a part of hard disk that is not used for file system. What is it and how is it used?

Ex. 5.11 — Discuss benefits of replacing mechanical hard disks by Solid State Drives in terms of memory hierarchy?

Ex. 5.12 — Discuss benefits of using mechanical hard disks over Solid State Drives?

Ex. 5.13 — What is the capacity of a hard disk with 64 sectors per track

with 128 tracks, if one sector is 128 bytes.

Ex. 5.14 — Discuss the benefits of setting the hard disk block size the same as the page size.

Ex. 5.15 — Consider the directory entry definition in Figure 5.7. What is pros and cons of this definition?

Ex. 5.16 — Consider the hard disk defined in Exercise 5.13. If we use the directory entry definition in Figure 5.7 and have the directory table with 100 of these entries (i.e., `MaxNumFiles = 100`), what is the largest file this file system can create assuming we use the contiguous allocation method? Hint: the size of a character is 1 byte; the size of boolean is 1 byte, and the size of an integer is 4 bytes.

Ex. 5.17 — Consider a system with a hard disk with 1024 tracks. Each track has 256 sectors. The disk sector size is 128 bytes. We want to use a FAT table to keep track of files and free-blocks. Each FAT table entry is 4 bytes. Calculate how many entries there are in the FAT table and how big the above FAT table is, in terms of memory requirement.

Ex. 5.18 — For the hard disk in Exercise 5.17, what is the largest size of file that can be created by this scheme? Assume the directory file size is 400bytes.

Ex. 5.19 — We are designing the i-node structure for the file system. We want to fit the I-node structure to one block (i.e, one sector). An I-node needs 7 words (i.e., 7×4 bytes) for file information such as the id of the file, size of the file, etc. Among all the pointers that exist in the i-node structure, we use two pointers for single-indirect block pointers and one pointer for double-indirect blocks. The rest are used for direct block pointers. What is the largest file size this i-node can create?

Ex. 5.20 — Study Log-based File System. Why would this file system be a good system for Solid State Drives?

Ex. 5.21 — What are the meta-data items to be considered in file system design?

Ex. 5.22 — What are the meta-data items to be considered in process/thread system design?

Ex. 5.23 — What are the meta-data items to be considered in memory management?

Bibliography

[1] Sedat Akyürek and Kenneth Salem. Adaptive block rearrangement. *ACM Trans. Comput. Syst.*, 13(2):89–121, May 1995.

[2] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[3] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Hystor: Making the best use of solid state drives in high performance storage systems. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 22–32, New York, NY, USA, 2011. ACM.

[4] Wayne A. Christopher, Steven J. Procter, and Thomas E. Anderson. The nachos instructional operating system. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 4–4, Berkeley, CA, USA, 1993. USENIX Association.

[5] Robert Geist and Stephen Daniel. A continuum of disk scheduling algorithms. *ACM Trans. Comput. Syst.*, 5(1):77–92, January 1987.

[6] Michael Kaminsky, George Savvides, David Mazieres, and M. Frans Kaashoek. Decentralized user authentication in a global file system. *SIGOPS Oper. Syst. Rev.*, 37(5):60–73, October 2003.

[7] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. *SIGOPS Oper. Syst. Rev.*, 31(5):238–251, October 1997.

[8] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2(3):181–197, August 1984.

[9] Steve Pate and Fred Van Den Bosch. *UNIX Filesystems: Evolution, Design and Impemenation*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[10] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.

[11] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Somesh Jha. A logic of file systems. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association.

[12] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[13] Toby J. Teorey and Tad B. Pinkerton. A comparative analysis of disk scheduling policies. *Commun. ACM*, 15(3):177–184, March 1972.

Index

Credits

1. Fig. 1.2: Copyright © 2008 by Canonical Ltd.
2. Fig. 1.6: Copyright © Chris Shrigley (CC by 2.5) at https://commons.wikimedia.org/wiki/File:IBM402plugboard.Shrigley.wireside.jpg.
3. Fig. 1.7: Copyright © Joe Mabel (CC BY-SA 3.0) at https://commons.wikimedia.org/wiki/File:LCM_-_IBM_029_Card_Punch_01.jpg.
4. Fig. 1.8: Copyright © Arnold Reinhold (CC BY-SA 3.0) at https://commons.wikimedia.org/wiki/File:Punched_card_program_deck.agr.jpg#file.
5. Fig. 1.9: Copyright © Mike Ross (CC BY-SA 3.0) at https://commons.wikimedia.org/wiki/File:IBM1442.corestore.jpg.
6. Fig. 1.10: Copyright © Erik Pitti (CC by 2.0) at https://commons.wikimedia.org/wiki/File:IBM_System_360_tape_drives.jpg.
7. Fig. 1.11: Copyright © Marcin Wichary (CC by 2.0) at https://commons.wikimedia.org/wiki/File:IBM_1401_lab.mw..jpg.
8. Fig. 1.12: Copyright © Todd Dailey (CC BY-SA 2.0) at https://commons.wikimedia.org/wiki/File:Altair_8800_at_the_Computer_History_Museum,_cropped.jpg.