

Operating System Principles

Joseph Pasquale

Department of Computer Science & Engineering
University of California, San Diego

Version 1.06

Copyright © Joseph Pasquale

Mar 12, 2022

Contents

1	Introduction	11
1.1	Why Study Operating Systems?	12
1.2	Operating System vs. Kernel	13
1.3	Purposes of an Operating System	15
1.4	Creating Desirable Illusions	17
1.5	Abstraction, Mechanism, and Policy	20
1.6	Overview of Topics	21
1.7	Summary	22
1.8	Exercises	22
2	Processes	27
2.1	The Process Abstraction	28
2.2	CPU and Memory Contexts	29
2.3	Process Memory Structure	30
2.4	Multiprogramming	33
2.5	Context Switching	34
2.6	Context Switching between Two Processes	38
2.7	Context Switching via the Kernel	59
2.8	Yielding Via the Kernel	61
2.9	The Process's Kernel Stack	62
2.10	Summary	64
2.11	Exercises	66
3	Timesharing	75
3.1	Quantum	76
3.2	Quantum Size	77
3.3	Process States	78
3.4	Process State Diagram	79

3.5	Logical vs. Physical Execution	82
3.6	Processes vs. the Kernel	83
3.7	When Does the Kernel Run?	84
3.8	Keeping Track of Processes	85
3.9	How Does the Kernel Get Control?	87
3.10	Preemption	88
3.11	User Mode vs. Kernel Mode	90
3.12	Context Switching in the Kernel	92
3.13	Entering the Kernel via the TRAP Instruction	93
3.14	Entering the Kernel via a Hardware Interrupt	95
3.15	The Big Picture	99
3.16	Threads	100
3.17	Kernel-level Threads vs. User-level Threads	105
3.18	Summary	111
3.19	Exercises	112
4	Scheduling	123
4.1	Characterizing Scheduling Performance	124
4.2	Longest First vs. Shortest First	127
4.3	Optimality of Shortest First Scheduling	130
4.4	First Come First Served	132
4.5	Round Robin	135
4.6	Shortest Process Next	137
4.7	Shortest Remaining Time	139
4.8	Multi-Level Feedback Queues	140
4.9	Priority Scheduling	143
4.10	Fair Share	145
4.11	Stride Scheduling	149
4.12	Real-Time Scheduling	154
4.13	Earliest Deadline First	158
4.14	Rate Monotonic Scheduling	160
4.15	Summary	166
4.16	Exercises	167
5	Synchronization	177
5.1	The Credit/Debit Problem	178
5.2	Critical Sections	181
5.3	Mutual Exclusion	183

5.4 Requirements for Mutual Exclusion	184
5.5 Peterson's Solution	192
5.6 Disabling Interrupts	196
5.7 Test-and-Set-Lock	198
5.8 Semaphores	205
5.9 Implementing Semaphores	214
5.10 Summary	219
5.11 Exercises	220
6 Interprocess Communication	229
6.1 Cooperating Process Structures	230
6.2 Inter-Process Communication	232
6.3 The Producer/Consumer Problem	233
6.4 Semaphores + Shared Variables	237
6.5 Monitors	244
6.6 Message Passing	254
6.7 Summary	264
6.8 Exercises	264
7 Deadlock	271
7.1 Resource Allocation Graphs	271
7.2 The Four Conditions for Deadlock	275
7.3 Deadlock Prevention	276
7.4 Deadlock Avoidance	281
7.5 Deadlock Detection and Recovery	289
7.6 Summary	291
7.7 Exercises	291
8 Memory Management	297
8.1 Process Memory Areas	297
8.2 Process Memory Layout	298
8.3 Role of the Compiler	299
8.4 Memory Space and CPU Time	301
8.5 Swapping	303
8.6 Memory Allocation Algorithms	305
8.7 First Fit Allocation	312
8.8 Best Fit Allocation	313
8.9 Worst Fit Allocation	315

8.10 Which Allocation Method is Best?	317
8.11 Fragmentation	318
8.12 Compaction vs. Breaking Memory Requests into Smaller Pieces	320
8.13 Memory Usage Analysis	325
8.14 The 50% Rule	326
8.15 The Unused Memory Rule	331
8.16 The Buddy System	333
8.17 Summary	346
8.18 Exercises	347
9 Logical Memory	353
9.1 Logical vs. Physical Address Spaces	354
9.2 Mapping Logical Address to Physical Addresses	354
9.3 Base and Bound Registers	358
9.4 Allocating Logical Memory Partitions	363
9.5 Any-Sized vs. Same-Sized Allocation	367
9.6 Segmented Memory	372
9.7 Paged Memory	397
9.8 Segmented Paged Memory	418
9.9 Cost of Translation	427
9.10 Summary	433
9.11 Exercises	435
10 Virtual Memory	443
10.1 Not Everything Needs to be in Memory	443
10.2 From a Logical Memory to a Virtual Memory	445
10.3 Page Faults	446
10.4 Faults in a Segmented Paged Memory	449
10.5 The Costs of Virtual Memory	454
10.6 Locality of Reference	455
10.7 Page Replacement	456
10.8 First In First Out Page Replacement	459
10.9 Optimal Page Replacement	463
10.10 Least Recently Used Page Replacement	467
10.11 FIFO vs. OPT vs. LRU	470
10.12 Approximating LRU	472
10.13 The CLOCK Algorithm	475
10.14 The Resident Set	494

10.15Local vs. Global Page Replacement Policies	495
10.16Multiprogramming Level	497
10.17Thrashing	499
10.18Swapping	500
10.19The Working Set	501
10.20Summary	508
10.21Exercises	510
11 File Systems	519
11.1 File System Characteristics	520
11.2 File Name Space	522
11.3 File Attributes and Operations	523
11.4 The Read/Write Model	524
11.5 The Memory-Mapped Model	532
11.6 Comparison of the Models	534
11.7 Access Control	536
11.8 File System Implementation	538
11.9 Goals	539
11.10Abstract Storage Device	541
11.11Summary	549
11.12Exercises	549



Acknowledgements

The author wishes to thank all the students in the Winter 2021 and Winter 2022 offerings of CSE 120 at UCSD for their constructive feedback. Special thanks to the following students who reported various typographical errors: Zhiqiang Pi, Lingyi Wang, Zhirui Dai, Yanling Huang, Ryan McClure, Luis Arroyo, Yunjin Chen, William Duan, Tiara Nguyen, Anirudh Perubotla, Pedro Suchite, Jason Vega, Sruthi Praveen Kumar Geetha, Xiaohan Fu, Nathan Sit, Christopher Harness, Jieun Lee, David Han Jun, and Hyunjo Lee.



Chapter 1

Introduction

This book is an introduction to operating system design, implementation and structure. We discuss principles that apply to *all* operating systems, rather than any specific operating system. Consequently, we do not focus on the particulars of, say, Windows or macOS, but rather we study those ideas that will apply to operating systems in general, including those particular systems. Knowing this material will help you to understand the specifics of any particular operating system, as they all share the basic elements discussed in this book. This chapter is an introduction to the book and an overview of the topics covered.

1.1 Why Study Operating Systems?

We begin with the basic question: Why study operating systems? First, as a computer scientist, it's good to know how your computer works. If something goes wrong with your system, knowing about how the operating system works will help you figure out what may be wrong. You'll also know, for example, how to enhance the system's performance if you know how the operating system works.

Second, system programmers get respect! It is generally recognized that, if you understand how an operating system works (at a level of technical depth that you will attain by studying this book), you can understand any type of complex software. Operating systems are typically one of the most, if not the most, complex types of programs that have ever been devised. So, if you know how to design and implement an operating system, you can probably tackle just about anything.

Third, operating systems is a classic area of computer science. If you are a computer scientist, you should know about this area. While even the earliest computers had operating systems, the area began its development in earnest in the early sixties. Indeed, it is often said that, during that most interesting time in the history of operating systems, most of the basic abstractions were determined, and that since that time, all we've been doing is "polishing a round ball." Not really fair: operating system design is continually changing, responding to an ever-increasing landscape of new devices and how

to incorporate them into a harmonious working system, and new demands of increasingly complex applications. Furthermore, the implementation of operating systems is also continually changing, given new developments in processor and memory architectures and their every-increasing capabilities. On the contrary, operating systems are very relevant in today’s world!

Finally, the subject of operating systems is intellectually very interesting and challenging. You’ll learn about forms of programming and program structure that you’ve likely never encountered. You’ll learn about how the operating system must anticipate new technologies so that they can be made available to users while the underlying system remains stable. All these issues put together present a very challenging discussion and one that is intellectually very stimulating.

1.2 Operating System vs. Kernel

Let’s begin by making a distinction between the concept of the *operating system* and that of the *kernel*. The term “operating system” has many interpretations. One is that the operating system is all the software that comes installed on your machine, minus the applications. It typically includes the graphical user interface (and many equate this interface with the operating system). In this book, which follows a typical university-level presentation of operating systems, our view of the operating system is made more precise, and is more limited. We focus on what is typically called the “kernel.” We

can define the kernel as that part of the operating system, broadly viewed, that all other programs depend on and that can be accessed via *system calls*.

The kernel works closely with the hardware: it is able to access hardware device registers and it is able to respond to hardware interrupts. No other software running on your machine, such as that for user applications, is generally allowed to do this. The kernel is also that part of the operating system that allocates basic resources, such as CPU time, memory space, and the use of I/O devices. From this point on, we will use the terms kernel and operating system interchangeably, knowing that by operating system, we really mean that part of it that we defined here as the kernel.

One cannot over-emphasize the importance and fundamentality of the operating system. A useful metaphor is that if we were to view our computer system as our universe, the operating system is what defines the “basic laws of physics” of this universe. It is the operating system that defines how time progresses and how space is distributed and structured. It is the operating system that determines what happens at various points in time, and what gets placed at different points in space. This is a very interesting and deep interpretation for what an operating system does, and in fact, this book is really about how it realizes these fundamental ideas.

1.3 Purposes of an Operating System

There are two purposes of an operating system. The first is to provide an abstract machine to the programmer. The features of this *abstract machine* – in terms of *functions* and *resources* – are what the programmer sees, and importantly, is *limited* to seeing. How these features are chosen (an interesting and complex subject in itself) can make all the difference as to whether an operating system is successful – whether programmers will want to use it, and whether it is able to accomplish what programmers desire – or not. The goals in the design of this abstract machine are *simplicity* and *convenience*. This is not surprising, as simplicity and convenience are all important to the user.

Before proceeding further, this is a good point to be more precise about who specifically is the “user.” In this book, our primary concern for how the operating system is used is for the *programmer* (in contrast to someone whose main use of the computer to read their email, browse the web, edit documents, etc.). This is important because it is the programmer who needs the ability to manipulate the machine to make it do what the application is being designed for. How the machine can be manipulated – what is possible and what is not – is determined by the operating system. Consequently, unless otherwise noted, when we say “user,” we especially mean the programmer.

The second purpose of an operating system is to manage resources. We will use the term “resource” a lot in this book. What is a resource? Simply,

for our purposes, a resource is anything that allows work to get done, as in something that provides functionality. For example, memory is a resource because it allows data to be stored and recalled. The CPU is a resource because it allows the processing of data. And to be even more precise, by memory being a resource, we mean memory *space*, and by the CPU, we mean CPU *time*. Note our reference to the universe metaphor; the operating system defines the rules for how space and time are defined and allocated.

The other term that appears in the second purpose and that we will use a lot is “manage.” By this, we simply mean how the resource is organized and maintained, generally to achieve a goal. For example, memory space may be organized as a sequence of blocks of bytes, which are then allocated as units and in some specific order. Similarly, CPU time may be organized as a sequence of intervals, which are then allocated as units and in some specific order.

The goals are typically performance, reliability, and security, either a subset or all of them simultaneously. Note that if there are multiple goals, they may actually conflict. For example, to achieve security, data may be encrypted, but this can degrade performance (i.e., it takes more processing time to encrypt, which contributes to delay, which harms performance). Or, to achieve reliability, multiple copies of files might be separately stored on different storage devices so that if any one of them failed, the file is still recoverable. But, this requires more storage space, which negatively affects efficiency, another form of performance degradation. How to strike the right

balance in achieving these multiple but conflicting goals of performance, reliability, and security, is one of the most interesting and hardest aspects of operating system design.

These two purposes of an operating system are made manifest in the interfaces and relationships between the operating system and

- the programs (what the programmer produces) that make use of the operating system, and
- the hardware that is managed by the operating system, as shown in Figure 1.1.

1.4 Creating Desirable Illusions

In a broad sense, we might say that the operating system *turns the undesirable into the desirable*. It turns the undesirable inconveniences of reality – the complexity of hardware, the fact that there are a single or limited number of processors, that there is a small or limited amount of memory, etc. – into desirable conveniences, which are actually illusions. These illusions consist of simple and easy to use resources, and that there are multiple, if not a seemingly unlimited number, of them, as in a very large number of CPUs and a large amount of memory, so large that we never need be concerned that we will run out. Of course, this can only be an illusion, as nothing is infinite. But that is precisely the view that the operating system is to provide to the

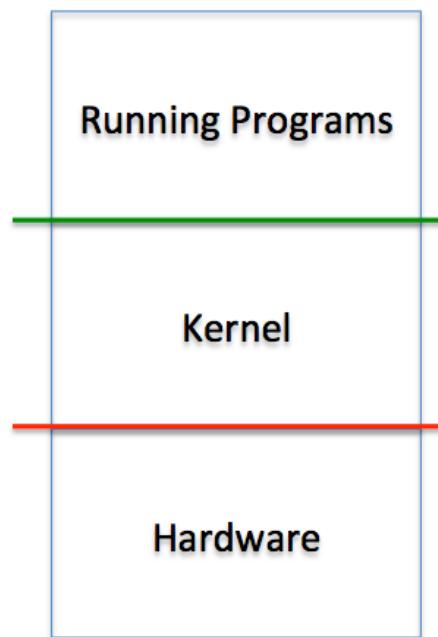


Figure 1.1: The operating system has two interfaces, each to satisfy the two purposes of an operating system: (1) the upper one, providing a simple and convenient abstract machine for programmers so they may design and implement their programs most effectively, and (2) the lower one, for managing the underlying hardware resources to achieve good performance, reliability, and security.

programmer. Consequently, the complex has been turned into the simple, and limits in numbers and amounts of resources are removed or masked. How this is achieved is part of the magic of operating system design.

From a programmer's point of view, we have to recognize that program design (e.g., what algorithms and data structures to use, and how to use them, to achieve the goal of the application) is hard enough! We want the programmer to focus their attention only on program design and not on how to make the machine run the algorithm (in an efficient, reliable, and secure, way – recall the goals of performance, reliability and security, which are *operating system* goals). We also want to minimize any imposition on the programmer in accounting for machine limitations, such as that there is a small number of processors or small amount of memory. Burdening the programmer with these issues introduces unnecessary complexity and may lead to the programmer modifying algorithms and data structures for reasons that have nothing to do with the application itself. This may also make the program not portable. Consequently, we, as operating system designers, would like the programmer to simply focus on what is hard enough, that of program design (and implementation), and leave all of the complexities of the machine and its management to the operating system.

1.5 Abstraction, Mechanism, and Policy

We now arrive at the three key ideas that form the most important part of this chapter. These ideas are *abstraction*, *mechanism*, and *policy*. We can define these by trying to answer the following “what/how/which” questions:

- Abstraction: *What* is the desired illusion, most relevant to the programmer?
- Mechanism: *How* is this illusion created (or *realized*, i.e., made real) using the basic functionality that is already provided.
- Policy: *Which* way should the mechanism be used, *to meet a particular goal*?

The latter two are often confused, and so it is helpful to consider the following basic difference between them. A mechanism is something that is fixed; it works one way and only that way. A policy, on the other hand, is something variable, or there may be many of them. Which policy is best suited for a system, a particular situation, will depend on the goals of the system. In fact, it is important to understand that in general, one cannot say that there is a *best* policy, as it depends on the goals of the system; for a different goal, a different policy is best. But regardless of the policy, the mechanism used to achieve the policy will be the same.

We will see these three key ideas appear over and over again when we discuss the various resources of an operating system throughout this book.

For each resource, we will discuss the abstraction or desired illusion that we are trying to achieve. We will then discuss what mechanisms are used to realize that abstraction. And finally we will discuss various policies, i.e., how to use the mechanism in different ways to meet a particular goal. As the goal varies, the policy will vary, but we will always be using the same mechanism, despite that it will be used in different ways. This will become clearer as we consider concrete examples of mechanisms and policies, as will happen in the next few chapters that focus on the CPU resource.

1.6 Overview of Topics

In this book, we will be discussing the following topics. We will be discussing various resources in terms of their abstractions, mechanisms, and policies. These resources include the CPU, memory, secondary storage (e.g., disks), and I/O (input/output). We will also be discussing operating system structure, by which would mean how the various parts or components of the operating system are related to each other, how data is exchanged between the various parts, and how control is transferred between the parts. These topics will make up the bulk of this book. We will also present some advanced topics, such as security, networks, and distributed systems.

1.7 Summary

In this chapter, we confronted the basic question: What is an operating system? We can answer by saying that it is that software that is an integral and indispensable part of the computer system. Its purpose is to make it easy for the user, especially the programmer, to use the system, and to keep the system running smoothly (to achieve a balance of the goals of performance, reliability, and security). In this book, we will be discussing fundamental aspects, i.e., *the principles*, of operating system design and implementation. We will be covering the resources of CPU, memory, storage, and I/O devices, and their abstractions, mechanisms, and policies. Finally, we will discuss some advanced topics, including security, networks, and distributed systems.

1.8 Exercises

At the end of each chapter, there is a list of exercises that you encouraged to do. Most can be done by a simple review of the chapter material, while some requires extra thinking. The exercises are graded using 0-3 stars according to their difficulty.

1. What is an operating system?
2. How does an operating system make the computer easier to use?*
3. What is an example of a difficulty a user would have in using a computer that had no operating system?**

4. What is the difference (that we use in this book) between a user and a programmer?*
5. What is meant by the term resource?*
6. How does a computer's operating system improve its capabilities?**
7. What is meant by a computer's efficiency?**
8. Why are speed and efficiency measures of performance?***
9. How is speed different from efficiency?**
10. What is meant by a computer's correctness?*
11. What does it mean for a computer system to be fault tolerant?*
12. Why are correctness and fault tolerance aspects of reliability?**
13. What is meant by each of the terms: privacy? authenticity? integrity?**
14. Why are privacy, authenticity, and integrity considered aspects of security?**
15. How is the operating system different from the kernel?
16. What is an example of something that might be part of the operating system that would not be part of the kernel?*

17. What are two actions a kernel can take that should not be allowed by a program, and why should they not be allowed?**
18. What does it mean to allocate a resource (given your earlier definition for resource)?*
19. What is meant by allocating CPU time?**
20. Why is meant by allocating memory space?**
21. What about memory time: what is meant by this, and does it make sense to allocate it?***
22. What about CPU space: what might be meant by this, and how might it be allocated?***
23. What is meant by the term abstract (as in abstract machine)?**
24. Why is the term interface relevant when discussing an abstract machine?**
25. Why are functions and resources relevant when discussing an abstract machine?**
26. What is a mechanism, and what is an example?*
27. What is a policy, and what is an example?*
28. Are mechanisms and policies aspects of an abstract machine: why or why not?***

29. In what way are the goals, simplicity and convenience, as a group (i.e., not individually, but when viewed as a group and its characteristics), different from the goals, performance, reliability, and security, again when viewed as a group?***



Chapter 2

Processes

The most basic function of an operating system is to allow a user to run a program. In fact, users want to be able to run multiple programs, at the same time, i.e., not have to quit one program when it is desired to run another one, and to allow this for many programs. How do we achieve this given a single CPU? For the purposes of this chapter, we will make the assumption that our computer only has a single CPU, and we will see that ultimately, we can assume this without loss of generality.

One of the goals of an operating system is to make it *seem* that there are many more CPUs than there actually are, i.e., despite the fact that there is only one or a limited number. In our development for how the illusion of creating many CPUs is achieved, we could choose our limited number to be something other than one, but whatever it is, it will still be a fixed and small number, such as four or eight. But we want to give our user the ability to

run many more programs at the same time than there are CPUs, no matter what that fixed number is. Consequently, we'll stick with the assumption that there's only *one* CPU, as we'll need to use the same mechanism to create many more beyond our fix number, regardless as to whether that fixed number is one or some other small number.

2.1 The Process Abstraction

This leads us to perhaps the most important abstraction that an operating system provides, that of a process. A *process* is an abstraction of a running program. What is especially important here is that a process refers to something that is *dynamic*. A common way of describing an activity (something that is dynamic) is to say that, at any point in time, the activity can be completely described by a set of state variables. How these state variables change over time defines the activity. This would be a more formal definition for a process, i.e., a description of all the state variables and their values at each point in time for the duration of the process. This is concisely captured in our informal definition of a process, a *program in execution*.

Consequently, a process has state that changes over time. Contrast this to a program, which is a *static* object. We can make an analogy to a kitchen recipe, which is a set of instructions used to cook something, and the act of cooking using the recipe. In this analogy, the recipe is a program (it is just a description, it is not itself an activity), whereas the act of cooking would be

the process (which is an activity, something dynamic, and can be described by how the state changes over time), e.g., what is the result of each step carried out so far in the recipe, what is the disposition of all the ingredients, what and how much of each are in bowls, what parts are being cooked and how long have they been cooked so far, etc., and finally, what is the next step in the recipe.

To support the process abstraction, we need resources. A *resource* is simply something that allows work to get done. In this case, to support a process, we need to provide use of the CPU to be able to execute instructions. We also need use of memory to maintain state, i.e., a description of the process at any point in time. A process may also require the use of I/O devices, which are also resources. Consequently, many resources need to be provided to realize the process abstraction. In this chapter, our focus is on the CPU resource (we will discuss memory and I/O in the chapters that follow).

2.2 CPU and Memory Contexts

We define the *context* of a process as all of the state that describes that process at a particular point in time. Since a process will require many resources, each such resource has its own state, including that of the CPU. The *CPU context* is defined as the values of all the CPU registers, especially the PC (program counter), SP (stack pointer), GPs (general purpose registers),

and any others that comprise the CPU.

A process will also require memory. The *memory context* includes the values of a set of pointers that describe which areas of memory contain the various parts of the process, including code, variables, and a stack of activation records. Finally there are other things that might make up the context of the process such as kernel-related state, but for the immediate purposes, we focus on the CPU context.

2.3 Process Memory Structure

Before proceeding to our discussion of the CPU context, we will need some background regarding the memory structure of a process. As shown in Figure 2.1, a process’s memory is generally composed of at least three areas: the *text area*, which contains the process’s code, i.e., program instructions; the *data area* that contains program variables, specifically ones that exist for the duration of the process, e.g., global variables and variables stored in dynamically allocated memory (commonly called the heap); and the *stack area*, which contains a sequence of activation records, one per pending procedure call.

The stack area is called a “stack” because it is managed as a last-in-first-out memory, which automatically grows when a procedure is called, and shrinks upon return. The activation record is a description of the state of the execution of the procedure, which only lasts between the time the procedure

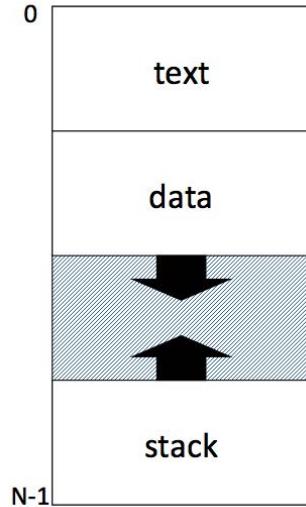


Figure 2.1: The memory areas of a process.

is called and when it returns. Since a procedure's local variables are defined only for the duration of the procedure call/return, they are located in the activation record, unlike global variables, which are stored in the process's data area.

The areas are laid out such that the text area occupies the area of memory starting at location 0, immediately followed by the data area, and at the opposite end of memory is where the stack is located. The data and stack areas are shown with boundaries that can grow and shrink. The data area grows when memory is dynamically allocated and shrinks when freed. The stack area grows when a procedure is called and thus an activation record is pushed on the stack, and shrinks upon a return, causing the activation record to be popped from the stack. As shown, the convention that we use in this

book is that memory addresses increase from top to bottom (as how the page of text you are reading is organized: you read from top to bottom). Many other books may use the opposite convention (high addresses are higher on the page) so take note.

For the stack, consider the contents of each activation record, as shown in Figure 2.2. Each activation record stores the state of a pending procedure, i.e., a procedure that has been called but has not yet returned. This state typically includes the address of where to return (which would be the point just after the call to the procedure), a link to the previous activation record, automatic variables, which are the local variables of a procedure that was just called, and possibly other data (such as register values). A dedicated hardware register, called stack pointer (SP), points to the top of the stack.

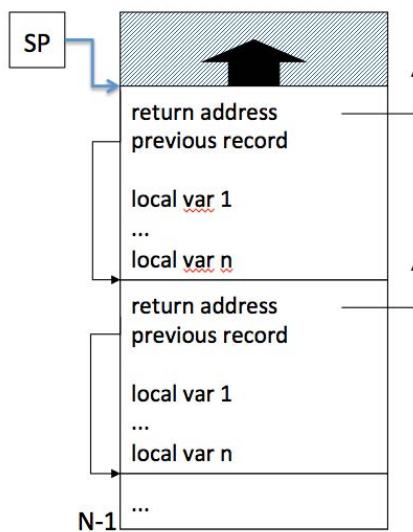


Figure 2.2: A stack area with multiple activation records.

At any point in time, the activation record that is most relevant is the uppermost one, which corresponds to the current procedure that the process is executing. Recall that there is one activation record per pending procedure call, and the topmost activation record is that of the most recent call. When that procedure returns, the topmost activation record is popped off the stack, and we know where the previous one is because of the link to the previous record contained in the activation record.

2.4 Multiprogramming

We're now ready to address our goal for supporting multiple processes, i.e., multiple running programs. A user would like to run multiple programs, such as a browser, editor, email client, calendar program, *simultaneously*. By simultaneously, we mean that all of these programs were started and have not yet completed; in other words, they are all processes that co-exist. However, note that not all of them are *actually* using the CPU; in fact, they can't if there are more processes than there are CPUs, and our assumption is that there is only one CPU, so only one (at most) can actually be running.

Note that most processes will typically not need the CPU at all times, as very often they will be paused, waiting for some input, such as a mouse to be clicked or keyboard character to be entered. As long as the process that can actually use the CPU is given it, the other processes can simply co-exist and not be affected. How do we take advantage of this to ultimately create the

illusion of multiple processes running simultaneously, when in reality, there is a single CPU?

To support this, we need to concept of *multiprogramming*. Given a process that is actually using the CPU, at some point, that process will generally need a resource, such as input from an I/O device. Say that that resource happens to be busy, and so the process cannot advance until the device becomes available to provide the process with input data. At that point, imagine that the process were to give up the CPU to another process. Given that there is a single CPU, there can only be one process actually running at any particular point in time, but if that process cannot use the CPU, as in our example, it might as well give it up and allow some other process to actually use the CPU.

This is accomplished with the procedure `yield(p)`, which when called allows process p to run, thus causing the currently running process to give up the CPU to process p. This requires the all-important mechanism that provides for context switching.

2.5 Context Switching

By *context switching*, we mean de-allocating the CPU from one process and allocating it to another process, the one that is being yielded to. We achieve this by first saving the CPU context (further uses of “context” in this chapter will assume it is the *CPU* context) of the currently running process, and

then restoring the context of the next process that is to run. Restoring the context of a process means obtaining context data that was previously saved to a predetermined area of memory and loading the CPU registers with that context data. This includes the general-purpose registers (GP), the stack pointer (SP), and last but not least, the program counter (PC).

It is imperative that the loading of the PC must be the very *last* instruction issued, as once the PC is loaded, program control is transferred to wherever the PC is pointing. The tricky part here is that, to perform the context switch, the CPU must actually be executing the context switch code, and when control goes to the next process to run, the context switch procedure is obviously no longer running. This is why it is critical that loading the PC is the very last instruction to execute. Thus, the context switch procedure does not “return” like a normal procedure as it simply jumps to the next-to-run, or *resumed*, process.

For the remainder of this chapter, we will focus on precisely how context switching is actually accomplished. Let’s look at a simple example of two processes, called A and B. Assume that A is currently running and is about to call `yield(B)` to give up the CPU to B. This now involves saving and restoring CPU registers, the GPs, the SP, and the PC (last!). Note that the PC, which the context switch mechanism is executing, is in the middle of the code for `yield`.

Figure 2.3 shows rudimentary code for the `yield` procedure, including some assembly language instructions indicated by “asm”. The function be-

gins with the setting of a local variable, called `magic` (of course it could be called anything, but we call it magic because of how it is used to accomplish what will seem like magic). We begin with setting `magic` to 0, and then saving the context of process A (assumed to be the currently running process). This includes saving the GPs, the SP, and the PC. When the assembly language instruction “save PC” is executed, what is the instruction that the PC is pointing to? The PC always points to the next instruction to execute, consequently, while “save PC” is executing, the PC points to the instruction corresponding to the “`if (magic == 1) return;`” statement (which will typically expand to multiple assembly language instructions, and so the PC will point to the first of these).

```

magic = 0;           // local variable
save A's context:   // current process
asm save GP;        // general purpose registers;
asm save SP;        // stack pointer
asm save PC;        // program counter, note value!
if (magic == 1) return;
else magic = 1;
restore B's context: // process being yielded to
asm restore GP;
asm restore SP;
asm restore PC;     // must be last!

```

Figure 2.3: Rudimentary code for `yield`.

The code does not show where in memory the register values are being saved to; we will consider that detail below. Suffice it to say that the register

values are saved to a predetermined area of memory, so that they can be recalled when the context of process A is to be restored.

After the register values are saved, we come to the conditional statement, if `magic` is equal to 1, return from `yield`. This may seem strange, given that `magic` was set to 0 a few instructions before, and its value has not changed during the interim. So why would we be checking whether `magic` is equal to 1? Let's go on to see why we need this.

Since `magic` is not equal to 1, the else clause executes, which now sets `magic` to 1. Next, the context of process B is restored by obtaining the values of process B's context (retrieved from a predetermined area of memory, not shown here) and are loaded into the various CPU registers, the GPs, the SP, and the PC (last!). Once the PC is loaded, process B actually begins running, because the PC (just loaded) is pointing to an instruction in the memory area of process B. At this point, `yield` is no longer running, and process B is now running. And so, a context switch from process A to process B is accomplished.

But, we're still left with the question: Why in the conditional statement was the value of `magic` checked as to whether it was 1, when it would seem that there is no way it could ever be 1? In fact, a typical “smart” compiler would “optimize” this code and remove that test, reasoning that it could never be true. This is exactly why, when operating system code is compiled, the optimizations are typically *turned off!* We (the operating system implementer) tell the compiler: “We don't need your help (thank you very much);

we know what we are doing, just translate the code as is!” Let’s take a closer look why that conditional statement is so important.

2.6 Context Switching between Two Processes

Figure 2.4 shows the program codes for two processes, process A and process B. They look very similar, but they are separate processes executing different programs (and they have their own separate memories). Process A begins by executing in its main procedure. Notice that it has a global variable called `me`. It also has a procedure called `yield` that will be called in `main`. Process B also has a `main` procedure, a global variable called `me` (which should not be confused with that of process A, as each has its own variable called `me`), and its own `yield` procedure (identical to that of A, but a separate copy as B is its own separate process) that will be called from within its `main` procedure.

These are two separate processes, with their *own separate memory areas*. The text areas of each one will have their respectively separate program codes. Their data areas have their own global variable, which happen to be called `me` in each case (they could have been called something else, and differently for each one). And finally, they each have their own stacks, separately keeping track of the activation records for each pending procedure, which will be different for A and B as they are separate processes that execute independently of each other. Where each one is in its execution history is independent of the other, and so separate stacks are needed.

```
int me;

main () /* process A */
{
    me = getpid ();
    yield (B);
    yield (A);
}

yield (p)
{   int magic;

    magic = 0;
    saveContext (me);
    if magic == 1 return;
    else magic = 1;
    restoreContext (p);
}

int me;

main () /* process B */
{
    me = getpid ();
    yield (A);
    yield (A);
}

yield (p)
{   int magic;

    magic = 0;
    saveContext (me);
    if magic == 1 return;
    else magic = 1;
    restoreContext (p);
}
```

Figure 2.4: Programs for two processes, A on the left and B on the right, each containing their own `main` and `yield` procedures, and their own global variables called `me`.

We will now go through a series of program steps, one by one to see exactly how a context switch is achieved between process A and process B. Figure 2.5 shows the memory areas of the two processes A and B, each containing their own text, data and stack areas. The text area for process A has its code as was shown in the previous Figure 2.4, its data area with global variable `me` that currently has no specific value because it has not been initialized yet, and finally its stack which will have the activation record for the `main` procedure when process A actually begins executing (i.e., by default, `main` is the first procedure that is “called”). Process B’s memory areas are set up in similar fashion. There’s a text area that has the code for process B, a data area that has the global variable `me`, and in this case, `me` is set to the value ‘B’.

As we will see, each process uses its own variable `me` to store its name, which it learns by calling the procedure “`getpid`.” (the code for `getpid` is not shown, only the call is shown and we can assume that it returns the name of the process that calls it). Given that `me` is set to “B” in process B, this implies that process B must have already executed in the past and had assigned the variable `me` with the value ‘B’. How this happened will become clear as we go on. Finally, process B has a stack that shows a top-most activation record that corresponds to the procedure `yield`, indicating it is the most recently pending procedure.

What we can deduce from all this is that, at some point in the past, process B ran, called `yield`, somehow gave up the CPU, and now process A

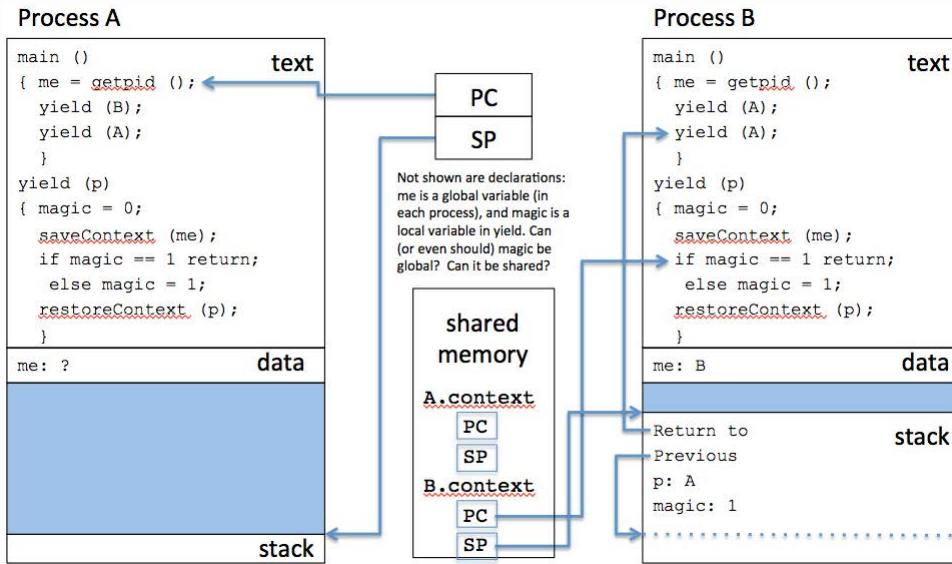


Figure 2.5: A is about to set me to its process ID, and yield to B. B had already yielded to A: note B's saved PC and SP.

is running. We know this because of the two hardware registers in the middle of Figure 2.4, which are the CPU's program counter (PC) and stack pointer (SP). Each, and PC in particular, is referencing an address in the memory of process A; hence, A must be currently running, i.e., using the CPU. The PC is pointing to the next instruction that process A is about to execute. The SP points to the top of the stack for A. Process A is about to set `me` (by calling `getpid`) and then yield to B (by calling `yield(B)`).

Process B had already yielded to process A some time in the past. We know this not just because, as mentioned earlier, the value of `me` (in B's data area) is already set to B, but because B's context, which contains a record of past values of the PC and SP, has been saved. This is shown in the lower

middle area of Figure 2.4 that is labeled “shared memory,” corresponding to memory that is somehow shared by both processes, something new we are now encountering.

Thus, in addition to the text, data, and stack areas for processes A and B, we now see that there is an additional memory area that is shared by both processes, which is where contexts are saved. Recall in our earlier discussion of how `yield` works, where we said that the contexts are saved to a “predetermined area of memory” but was not further discussed; we now see where this is. In this memory area, we find saved values for the context of A and the context of B. Currently, the saved context of B shows values of the PC and SP saved when B was last running, and the memory addresses that they are pointing to within the memory area of process B.

Figure 2.6 shows that process A just finished executing the statement “`me = getpid()`.” Process A has just set `me` to ‘A’ and is about call `yield(B)`, to give up the CPU to process B. Note that the PC register (in the upper middle of Figure 2.6) always points to the next instruction to be executed, in this case a call to `yield` with parameter B. Note that the global variable `me` was set to ‘A’ after the execution of that first statement, a call to `getpid()` which is a procedure that returns the process name (i.e., identifier or “ID”) of the process that is calling `getpid`, i.e., the currently running process. Consequently, process A learned of its name by having called `getpid` and assigned it to the global variable `me` which we see in A’s data area.

In Figure 2.7, we see that process A has entered the `yield` procedure

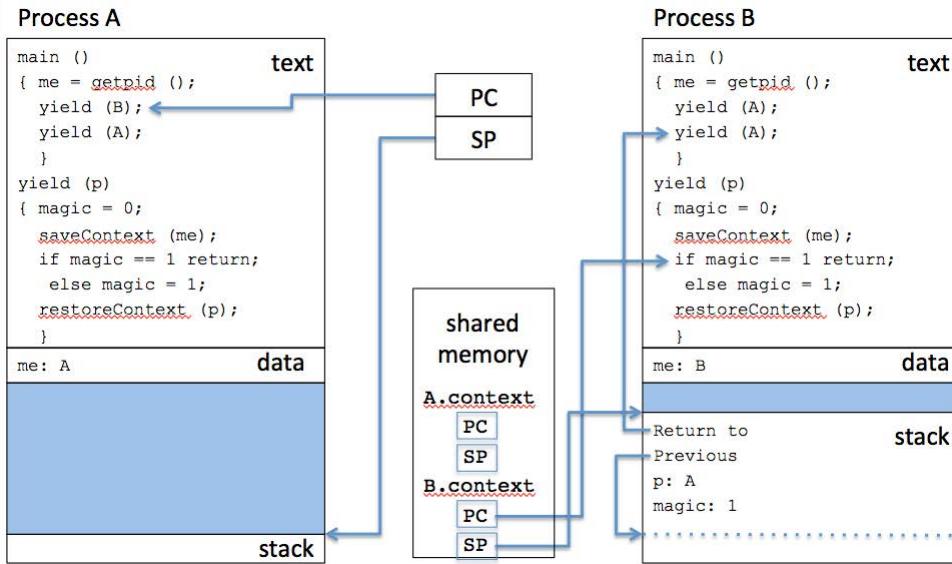


Figure 2.6: Process A has just set me to ‘A’ and is about to call `yield`. The PC always points to the next instruction to be executed.

and so an activation record is pushed onto the stack. It contains links and local variables `p` and `magic`. The “Return to” link points to where process A should return to when this (first) call to `yield` completes. The ‘Previous’ link points to the previous activation record, so that it is known how to remove the topmost activation record when the call to `yield` returns. The local variable `p` is set to the value ‘B’ because this was the value passed as a parameter to `yield`. Finally, there is the local variable `magic`, which is currently unassigned.

Notice that the PC is currently pointing to the first instruction of the first statement in `yield`, which has not executed yet, and the SP is pointing to the top of the stack, which is where the activation record for the currently

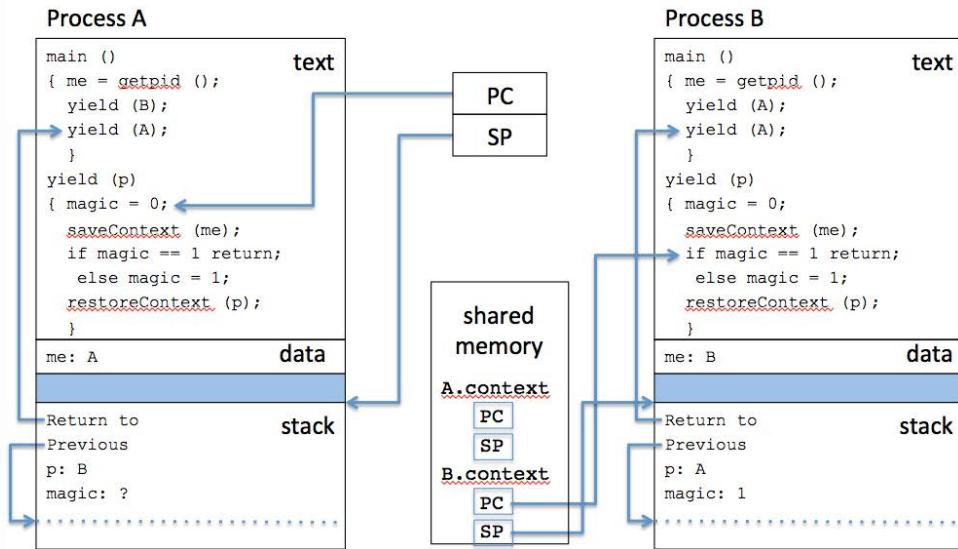


Figure 2.7: Upon entering `yield`, an activation record is pushed on the stack. It contains links, and local variables `p` and `magic`.

executing `yield` procedure is.

Next, as shown in Figure 2.8, the statement “`magic = 0`” has just been executed and so the variable `magic` is set to 0. Note that the value is recorded in the activation record for `yield` in process A. The variable `magic` is called an automatic variable because it is automatically allocated when the `yield` procedure is called. There is no need to explicitly (i.e., via the program itself) allocate memory for this variable, as it is done automatically because the compiler will have included instructions for the stack to grow and include the activation record for `yield`, and since that activation record includes memory for `magic`, the memory is thus “automatically” allocated. The PC is pointing to the next instruction to be executed, which is to save the context

of process A.

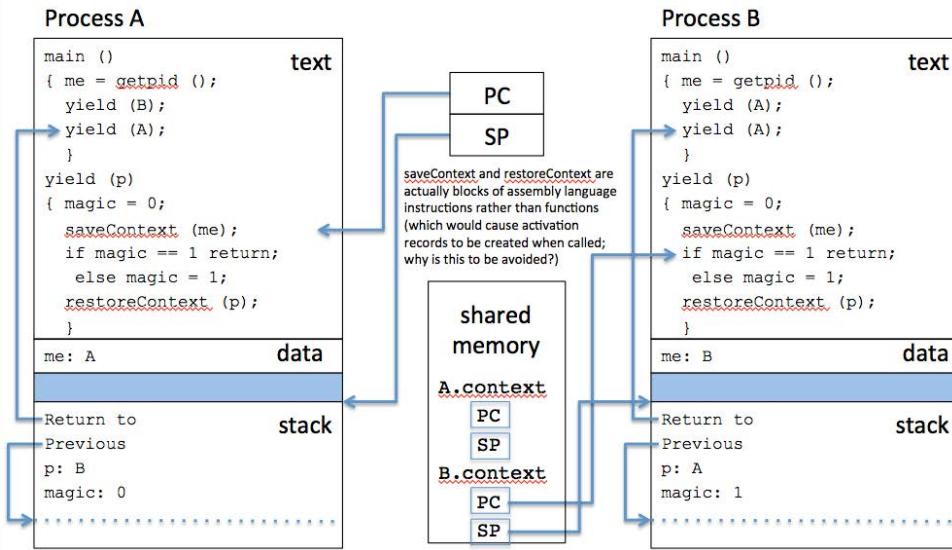


Figure 2.8: Variable `magic` is set to 0. It is an automatic variable, dynamically allocated on the stack. Next: save context.

Figure 2.9 shows that the context of process A was just saved. This is the result of having executed “`saveContext(me)`”, which is actually a macro (and not a procedure call – why is this?) for the three instructions shown in Figure 2.3 that save the GPs, the SP, and the PC. We know that it is the context of A because of the reference to the variable `me` in the “call” to `saveContext`; the macro will simply use the variable `me` to determine where to save the context. From Figure 2.9 we can see that these registers are saved in “`A.context`” in the shared memory area. Later on, if and when process A is to be resumed and its context is to be loaded, the “predetermined” memory area where the context for A can be found is here. Not shown (for brevity)

are the GP registers, which are also part of the process's CPU context; they would also be saved at this point.

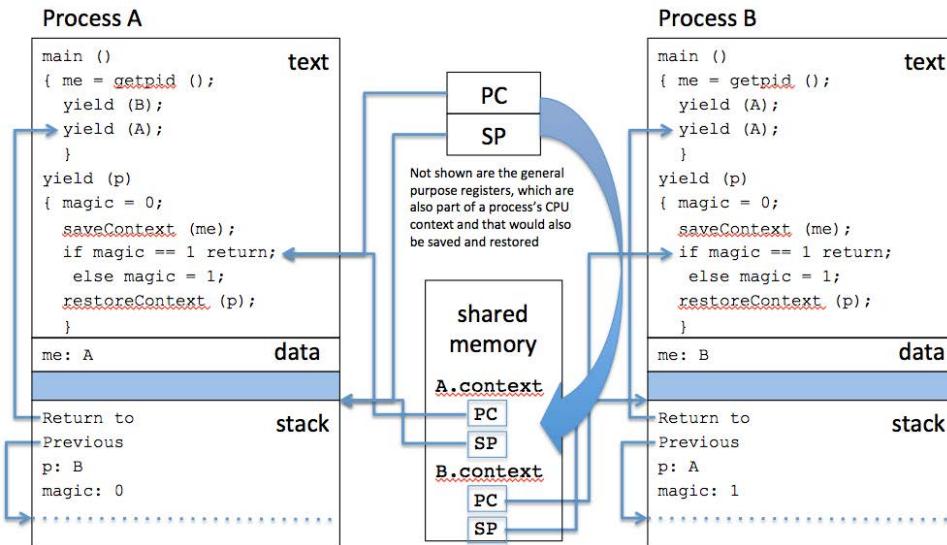


Figure 2.9: A's context is now saved. The saved PC points to just after the call to `saveContext`. Compare this to B's saved context.

The context of process A is now saved. The saved PC points to the instruction that comes right after the last instruction in the `saveContext` macro, which would be the first instruction of the conditional statement that checks whether `magic` is equal to 1.

Compare the saved context of process A to the saved context of process B (that was saved at some point in the past). They are very similar. In each case, the saved PC value points to the conditional statement that checks whether `magic` is equal to 1, and the saved SP value points to the top of the respective stack that contains the activation records for the corresponding

pending `yield` procedure call. This will become important as we proceed.

In Figure 2.10, process A just checked whether `magic` is equal to 1, but since `magic` is equal to 0, the conditional test fails. Consequently, the PC points to the else clause, which is about to set `magic` equal to 1.

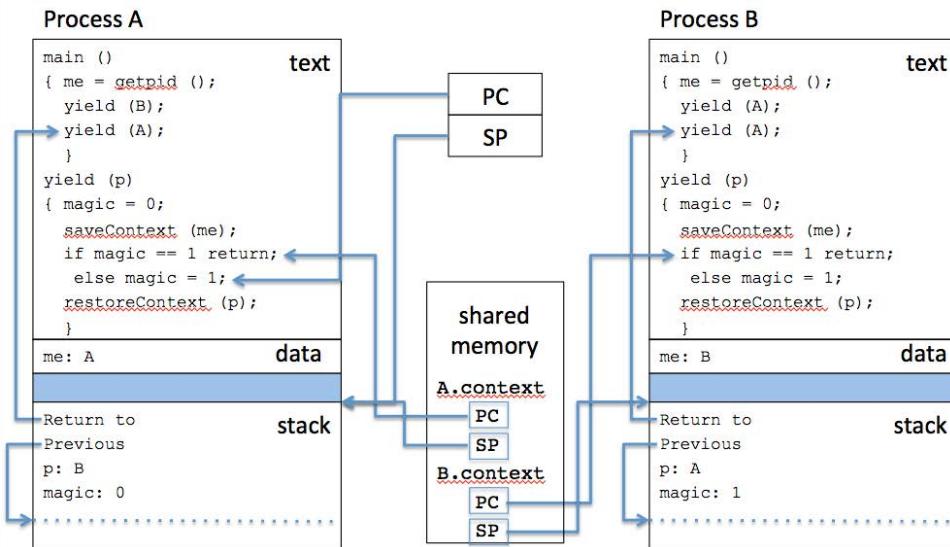


Figure 2.10: Process A just checked whether `magic` equals 1, which was false, and so, on to the else clause to set `magic` to 1.

In Figure 2.11, process A just set the value of `magic` to 1, and is about to restore the context of process B. This is just like process B's situation in the past when it restored A's context, which must have happened just as we've observed for process A because the `yield` procedures are exactly the same for both process A and process B.

In Figure 2.12, process B's context is restored, and so the machine state (as shown by the contents of the registers SP and PC) is now that of process

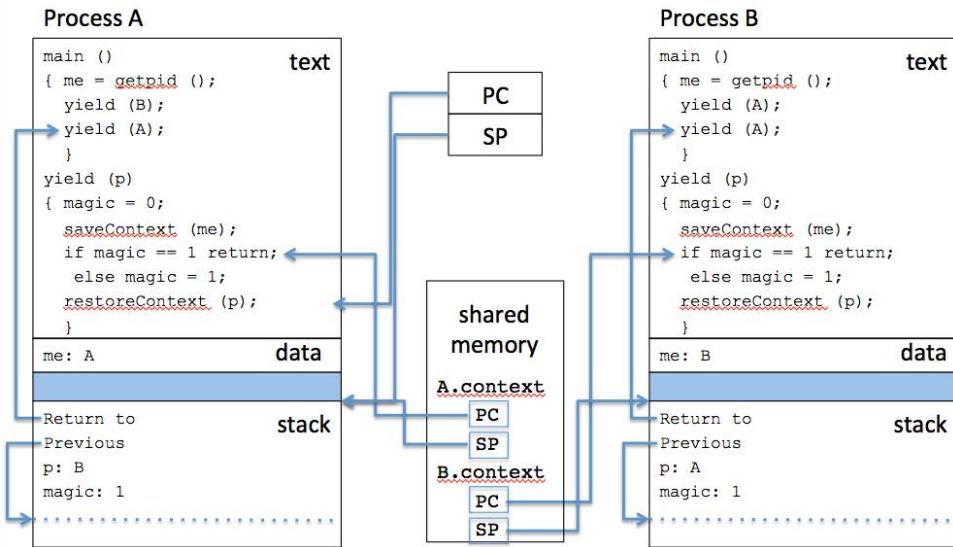


Figure 2.11: Process A sets `magic` to 1, and is about to restore B's context (just like B's situation in the past when it restored A's context).

B. Previously saved values of B's context in the shared memory area have been copied to the registers SP and PC, in that order. When the PC is loaded, the CPU begins executing wherever the PC points to, which in this case happens to be the conditional statement in process B that will check whether `magic` is equal to 1.

In Figure 2.13, process B is now executing – a context switch was achieved from A to B! Since `magic` equals 1, process B returns from `yield`, unlike the last time when `magic` was equal to 0. How was it that `magic` was set to 1? We can see that, in the past, it must have been set to 1 because that was the last thing that process B did before it gave up the CPU. No wonder we named it *magic*!

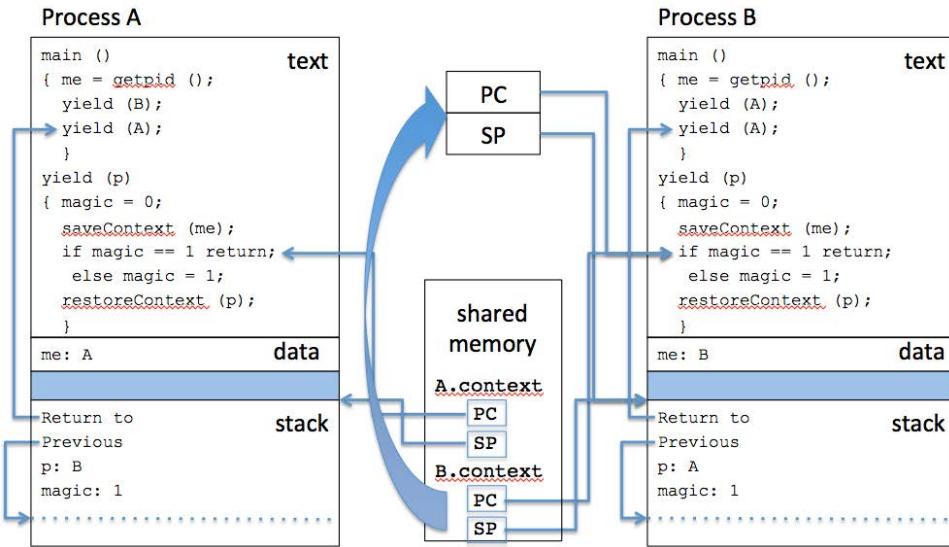


Figure 2.12: Process B's context is restored (i.e., the machine state is now that of B). The PC points to the if (conditional) statement.

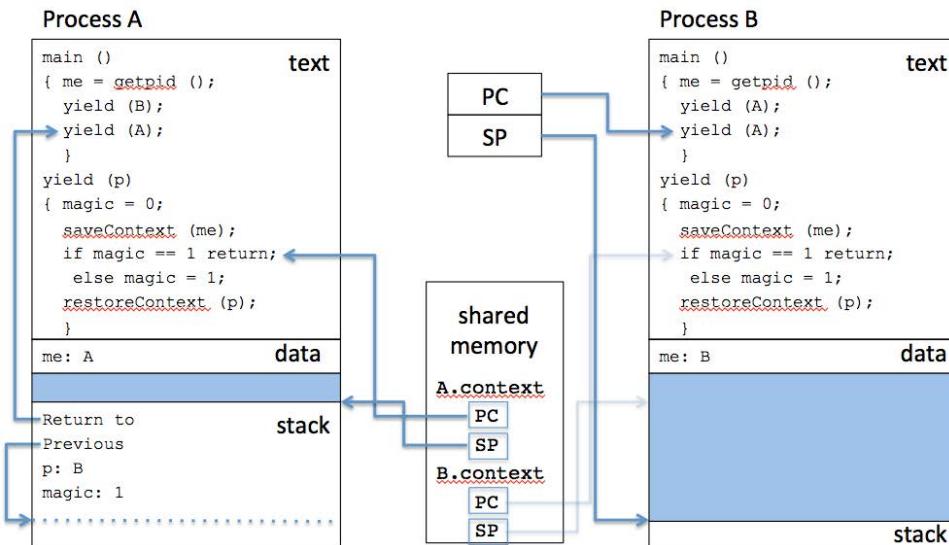


Figure 2.13: Since magic equals 1, B returns from yield (unlike last time when magic equaled 0).

Process B is now about to call `yield` (the second call to `yield` in the code for process B), to give up the CPU to process A (in our example, we did not see the execution of the first call to `yield` by process B; all we saw is that the first call to `yield` had to have taken place in the past). By the way, if you are wondering why would process B want to yield to process A again, there is no good reason, so think no more. This is simply a contrived example to demonstrate how `yield` works.

In Figure 2.14, we see that process B is now in `yield`. Just as we saw with process A, the `yield` procedure executes, and the first thing that will happen is that `magic` will be set to 0, but this time, we're looking at the execution of process B, not A. You should be able to figure out what happens next, so take a moment to try and do so. You can then read on and see if you got it.

In Figure 2.15, we see that `magic` was set to 0, and the `saveContext` macro is about to be called with parameter `me`, currently set to 'B', as can be seen in the data area of process B. This means that it is process B's context that will be saved, and so the values of the PC and SP will be saved to the shared memory area that contains the saved values of process B's context.

In Figure 2.16, the contents of the SP and PC registers are saved, with the PC pointing to the conditional statement to check whether `magic` is equal to 1.

In Figure 2.17, we see that the conditional statement failed, because `magic` is equal to 0. The reason for this is that `magic` was just initialized to 0 (when process B resumed executing), and so the else clause will execute

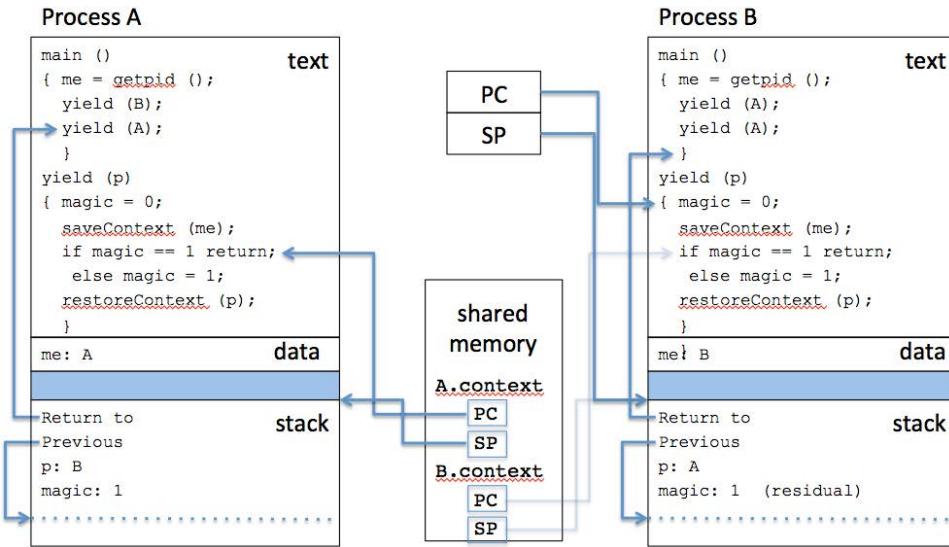


Figure 2.14: Process B just called yield and is about to execute the first statement in yield, to set `magic` equal to 0. We've seen something similar to this before!

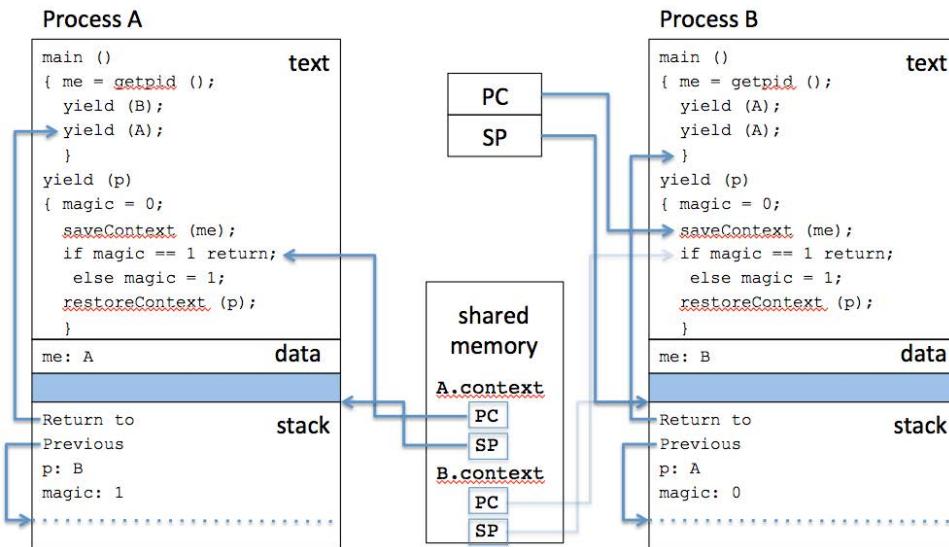


Figure 2.15: Process B is yielding to process A.

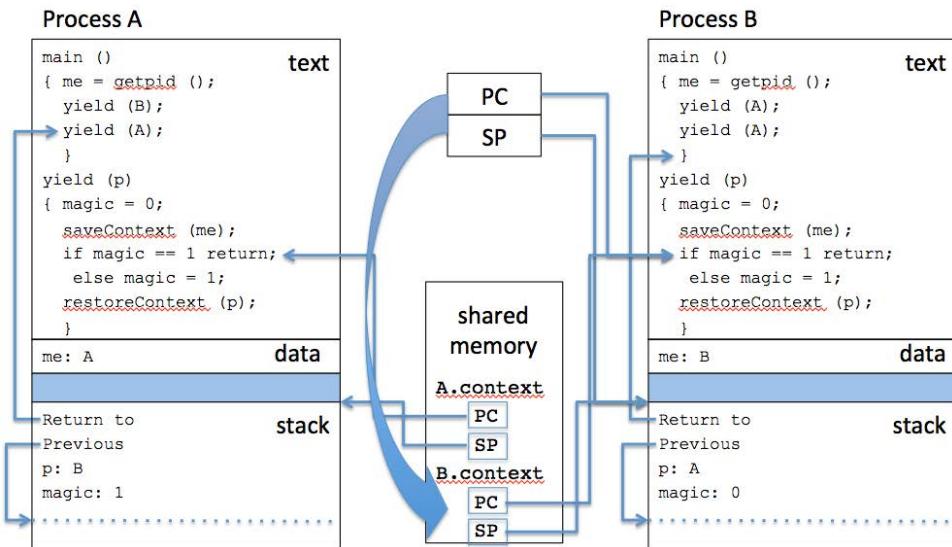


Figure 2.16: Goes through the same steps that we saw before.

next as indicated by the PC.

In Figure 2.18, `magic` is now equal to 1, and we're about to restore the context of process A, because of the parameter `p` is equal to the value 'A', as can be seen in the activation record on process B's stack.

In Figure 2.19, process A will now resume because the values of process A's context, which were saved before (and which we saw how they were saved) are loaded into the SP and PC registers. Again, it is very important that the PC be loaded last, because once it is loaded, process A begins executing. And where will it begin executing? The execution will resume at the address to which the PC is pointing, which is the conditional statement that checks whether `magic` equals 1.

In Figure 2.20, process A has just returned from `yield` and is now about

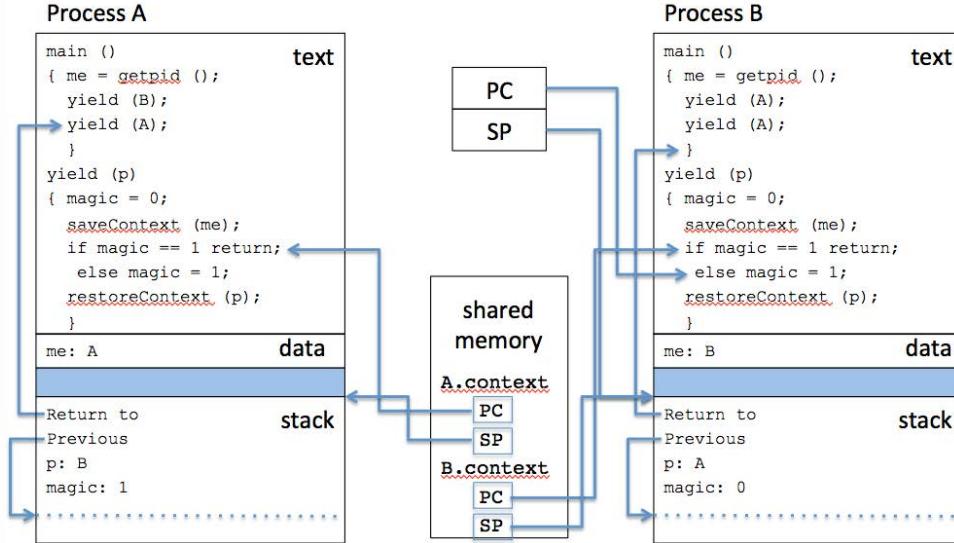


Figure 2.17: Note the value of textttmagic.

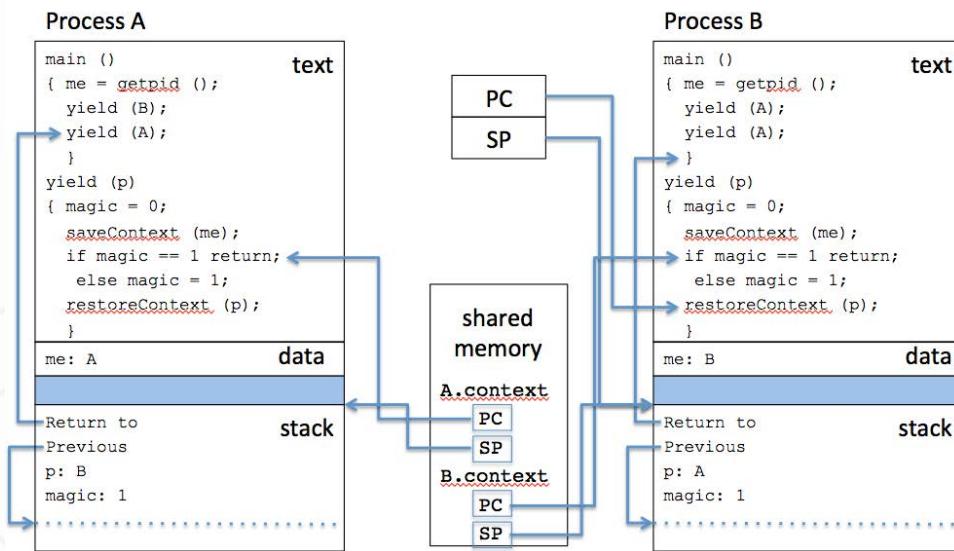


Figure 2.18: About to restore the context of process A.

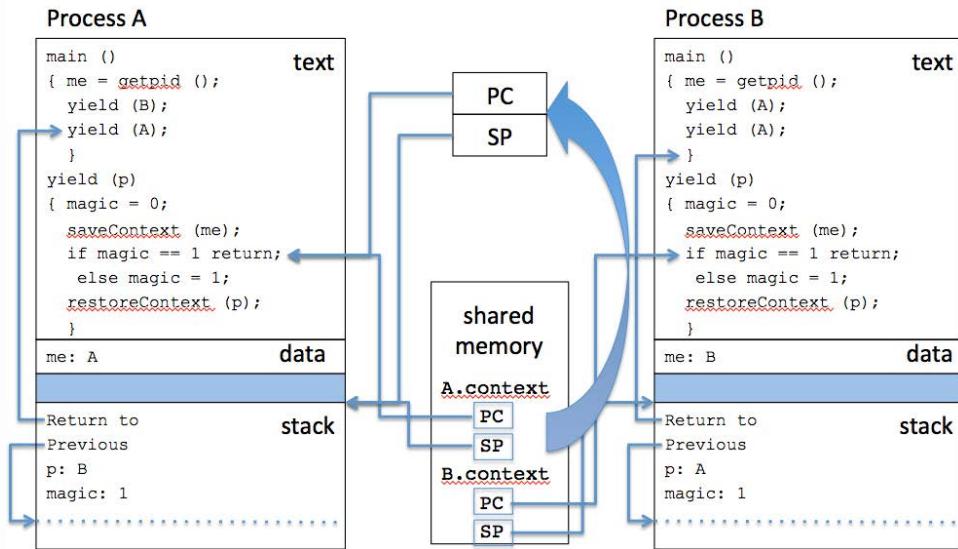


Figure 2.19: Process A resumes. Note the value of `magic`.

to call `yield` again, but this time to itself! If our context switching mechanism works properly, there should be no problem with a process yielding to itself. Let's see if it actually works.

In Figure 2.21, process A is now executing in `yield`, about to set `magic` to 0. In Figure 2.22, we see that `magic` was set to 0, and now the `saveContext` macro is about to be executed, with the variable `me` set to 'A' (as shown in the data area). Consequently, as we saw before, process A's context will be saved, which is what is done by `saveContext`. In Figure 2.23, we see that the context of process A was indeed saved, and note that the value of `magic` is equal to 0.

Consequently, the next statement to be executed, which is to check whether `magic` is equal to 1, will fail. We should now have a good understanding of

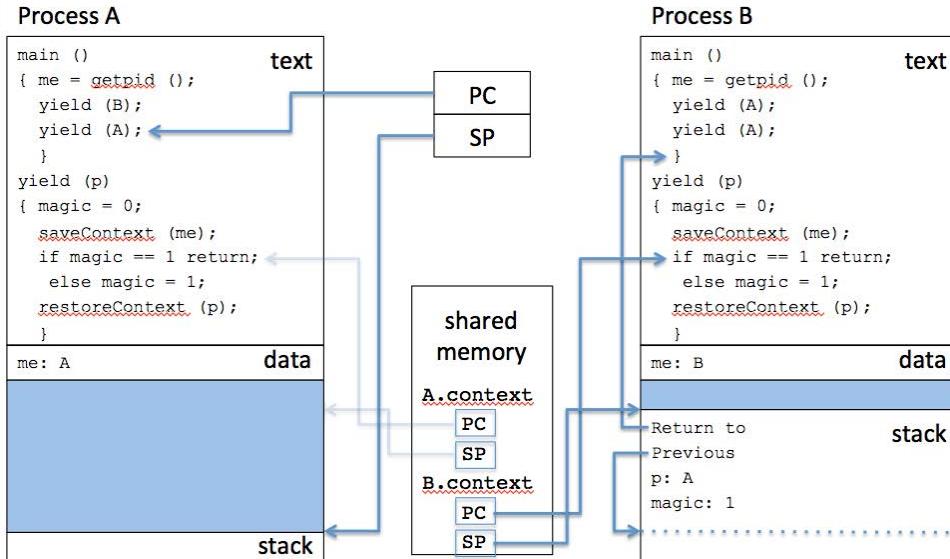


Figure 2.20: Process A returns from `yield`, and now is about to call `yield` again, but to itself!

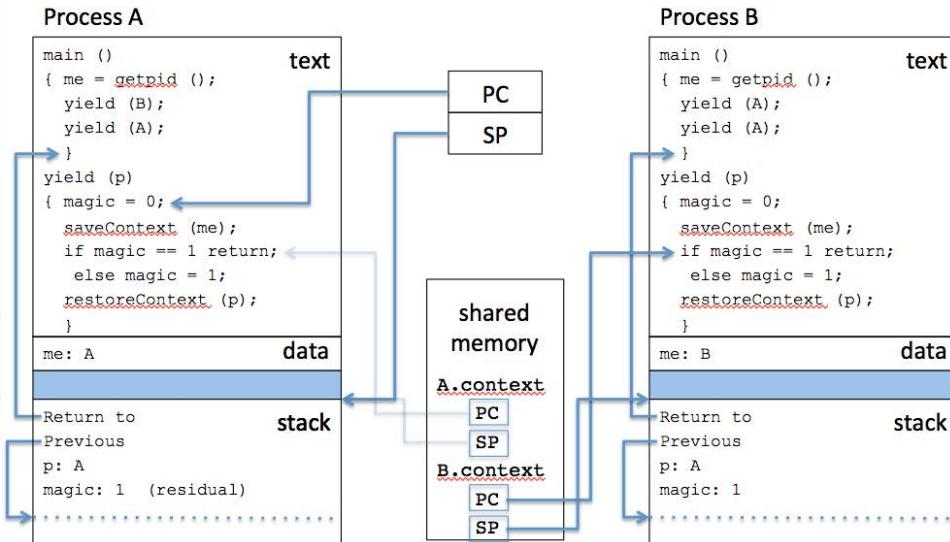


Figure 2.21: Process A is now in `yield`. Note the new activation record on A's stack.

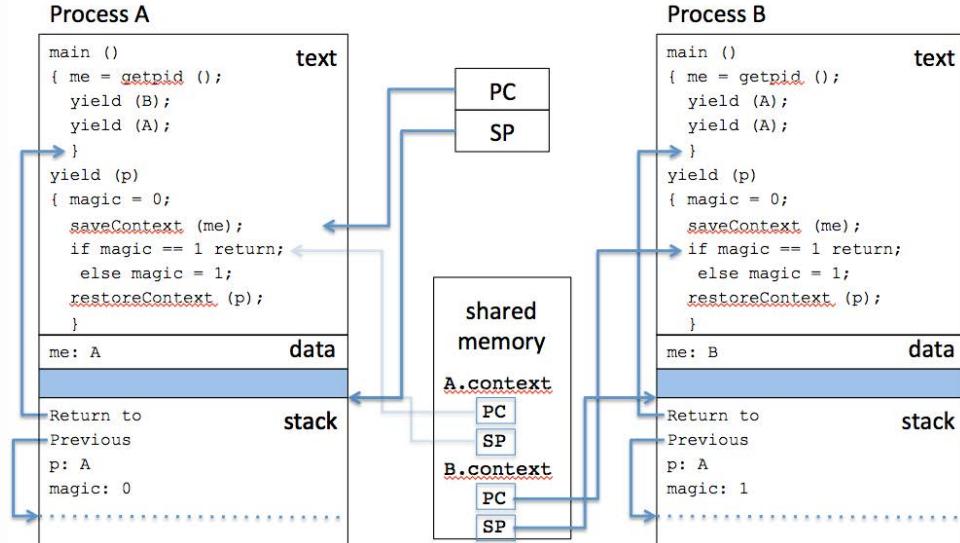
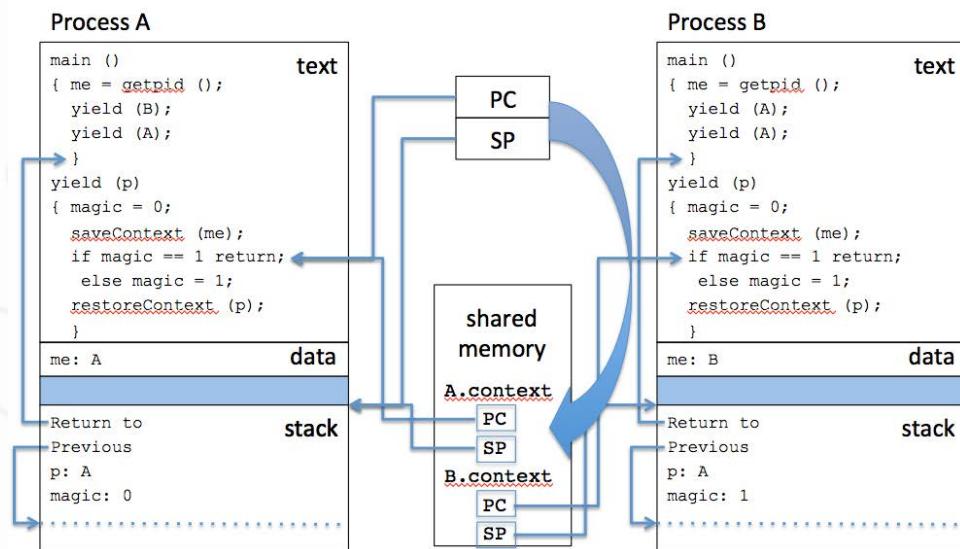


Figure 2.22: About to save the context of process A.

Figure 2.23: Note the value of `magic`.

what happens next.

In Figure 2.24, we see that the test of the conditional statement failed, and that `magic` is now about to be set to 1.

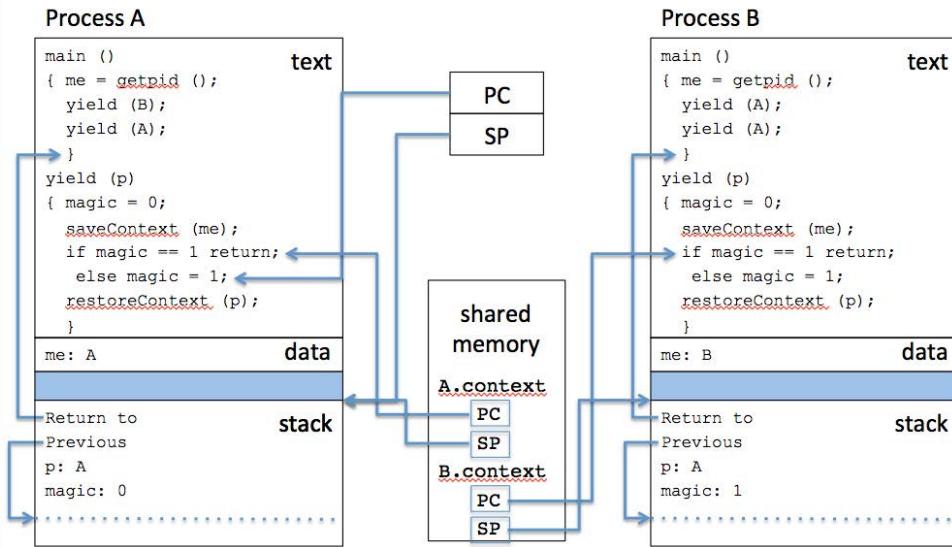


Figure 2.24: Set the value of `magic` to 1.

In Figure 2.25, the context of the process referred to by variable `p` is about to be restored, and variable `p` is equal to ‘A’, as we could see in the activation record for process A. This tells us that the context of process A is about to be restored. The context of A was just saved, and whatever was saved will now be restored.

In Figure 2.26, the context of A is restored. Note where process A is now executing. It is about to check whether `magic` is equal to 1, because that is where the PC is pointing. Indeed, `magic` is equal to 1.

In Figure 2.27, we see that `yield` returns, and the activation record for

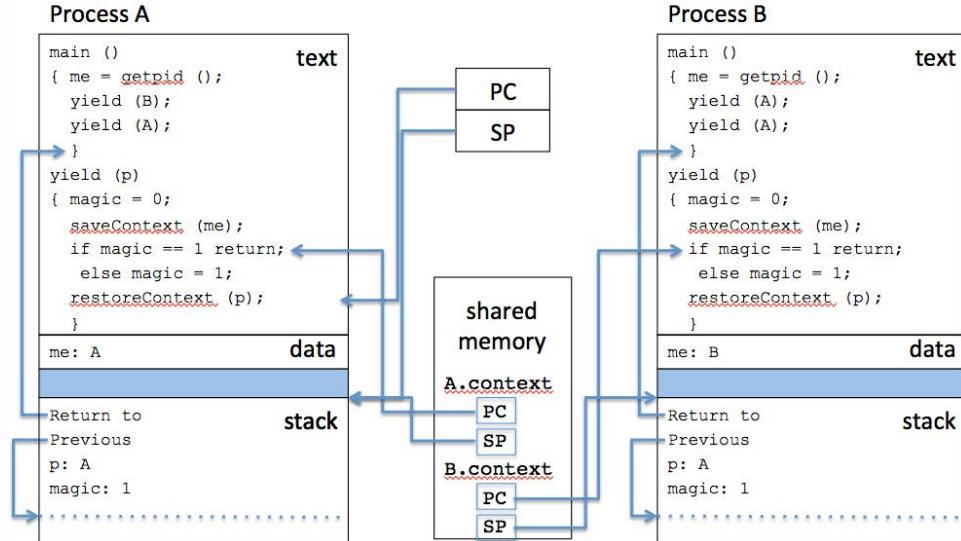


Figure 2.25: Restore the context of A!

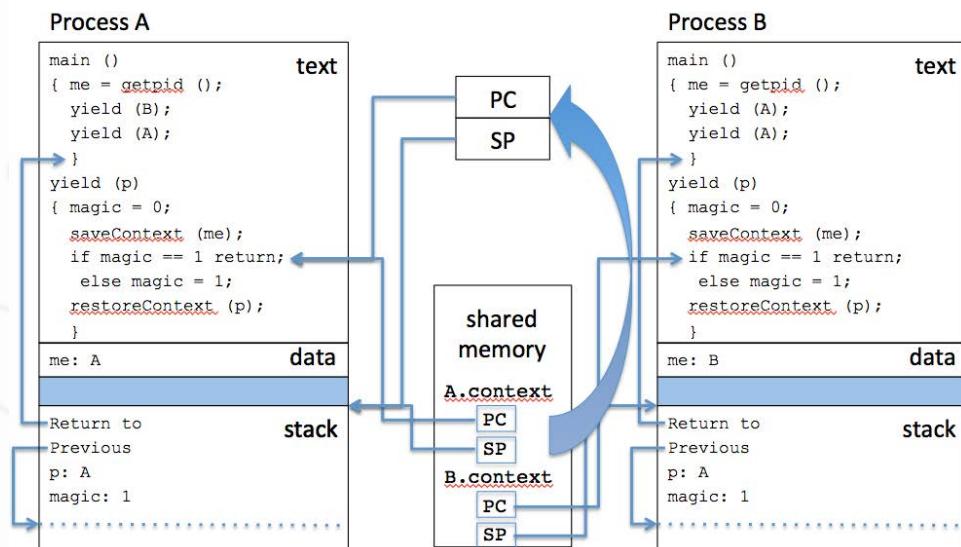


Figure 2.26: Note where process A executes.

`yield` is removed, which simply means that the stack pointer is set to the previous activation record (which is that of main). Process A achieved a yield to itself. Works like magic.

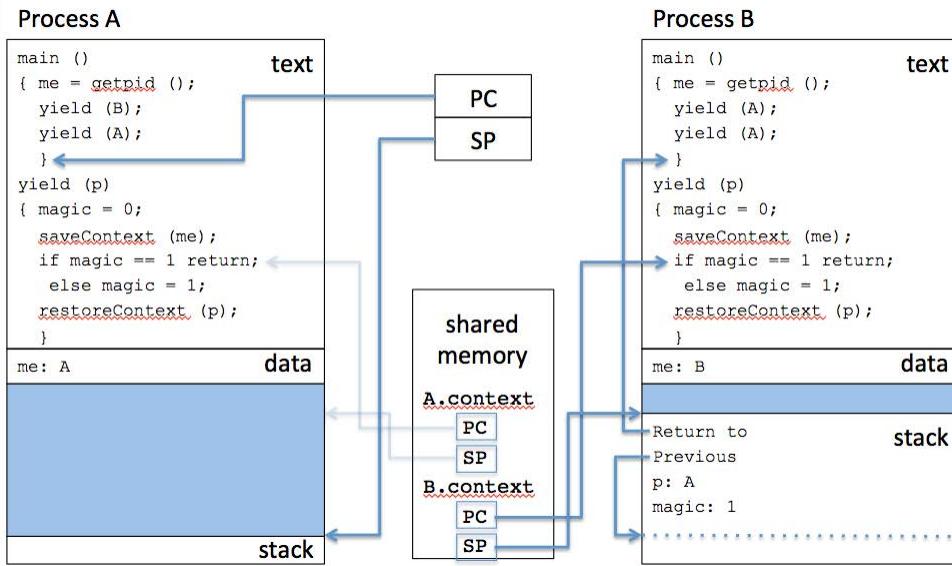


Figure 2.27: Return from yield.

2.7 Context Switching via the Kernel

In the example just presented, each of the two processes gave up the CPU to each other, and the mechanism that achieved this transfer of control is called context switching. However, we have yet to talk about the kernel! We are now ready to see where the kernel fits within this scenario.

The memory that contains the code for `yield` as well as the memory for the saved process contexts, the latter of which was in an area that we called

“shared memory,” are actually *part of* the kernel. They are a part of the kernel for a number of reasons, which we will see in the next few chapters, one of them being for protection, as the memory of the kernel is automatically protected from being accessed by the code of processes. As for the saved process contexts, they are only needed by the `yield` routine anyway, and since `yield` is in the kernel, it makes sense that the saved process contexts are also in the kernel.

But what is the kernel? *The kernel is code that supports processes.* But why is the kernel itself not a process? Since we defined a process as “a program in execution,” doesn’t this mean that when the kernel is executing, it is a process? That would certainly be one possible and valid interpretation. An alternative and more commonly held interpretation, and the one that we use in this book, is that the kernel is not a separate process, but rather *an extension* of existing processes.

The kernel contains a set of procedures, some of which can be called by the currently running processes (an example that we saw earlier is `yield`), and so by allowing a process to call kernel code, the kernel acts as an extension of the current process. However, there is a distinction between existing processes and the kernel: the kernel is special in that *it has its own text, data, and stack areas.* In fact, it has multiple stacks, one per existing process.

2.8 Yielding Via the Kernel

Let's go through our context-switch example, but this time we will see how yielding occurs *via the kernel*. In Figure 2.28, we again see two processes A and B, and A is about to yield to B. When a program for A is compiled, and the compiler sees the call to `yield`, it replaces it with instructions that will include a special instruction, called the `TRAP` instruction. When the `TRAP` instruction is executed, it causes control to go to the kernel.

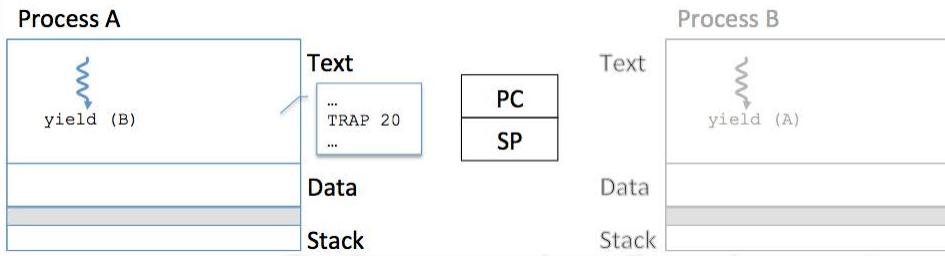


Figure 2.28: Yielding via the kernel.

In Figure 2.29, we see that there is a memory area that we have not shown before, that we call *kernel space*. Kernel space is distinguished from user space, which is what we call the memory areas for all processes that we have already seen. We can now see why we interpret the kernel as an extension of the process, because when process A calls `yield`, which causes the `TRAP` instruction to be executed, the process jumps into this separate kernel space, and the kernel space is effectively an extension of process A (while process A is actually running).

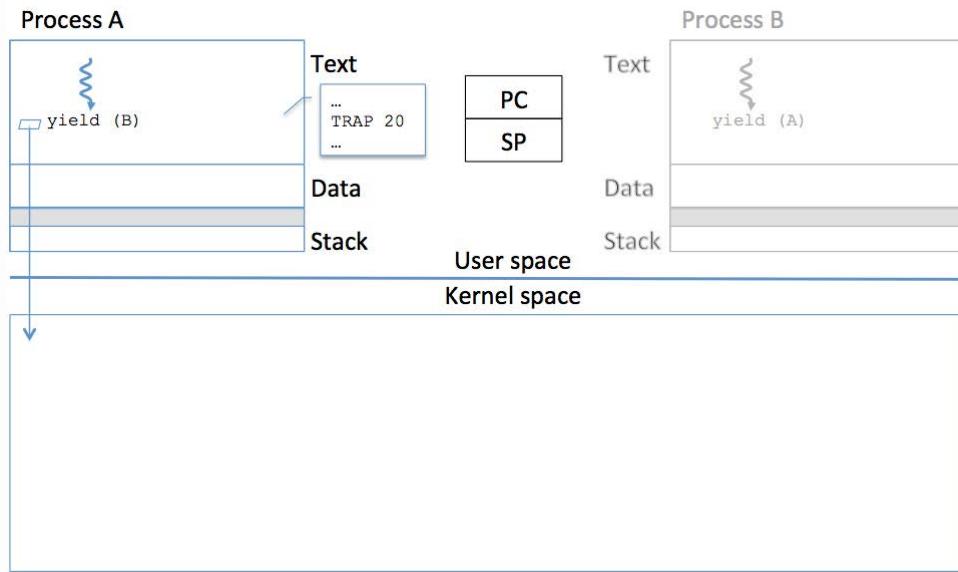


Figure 2.29: TRAP causes kernel entry.

2.9 The Process's Kernel Stack

The code for `yield`, as well as the process contexts that get saved, are all stored in the kernel space. This is shown in Figure 2.30. The kernel text area is where we find the code for `yield`, and the kernel data area is where we find the save contexts for all processes. In addition, there are a number of kernel stacks, one per process. These should not be confused with the stacks that belong to each process in the user space area.

The basic distinction is as follows. When the process is running its own code, which is in its text area in user space, the stack in user space is used to store activation records for pending procedures. When a process jumps into the kernel via the `TRAP` instruction and executes kernel code, the stack in

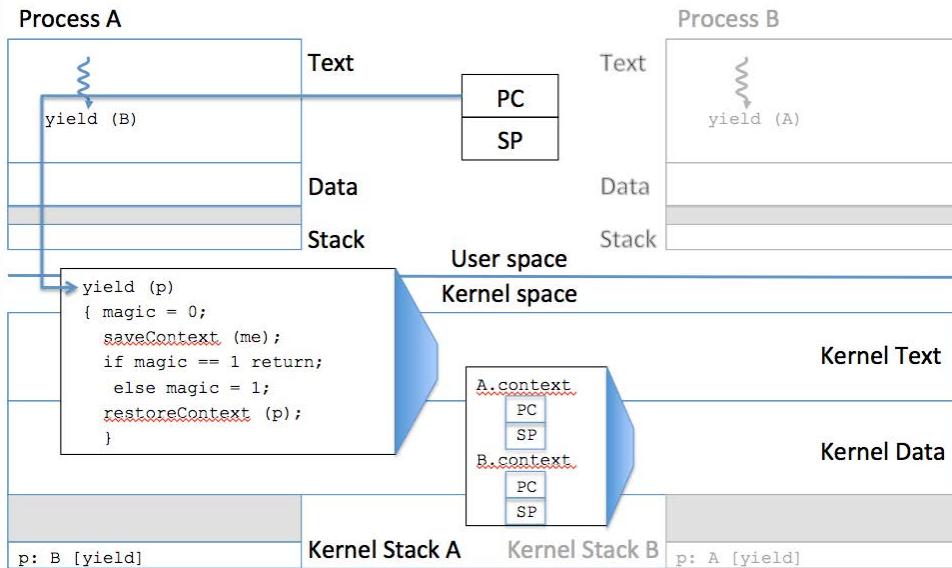


Figure 2.30: `yield` and saved contexts are in the kernel.

the kernel area is used that is associated with that process. While executing in the kernel, any calls to procedures (in the kernel) will cause activation records to be stored in the kernel stack. The reason for separating these two stacks will become clear as we go on.

In Figure 2.31, we now see the execution of `yield`, which is occurring inside the kernel. Just as we observed earlier, when `saveContext` is executed, the contents of the SP and PC registers are copied into the area where saved process contexts are stored. But unlike before, we now see that this area is in the kernel data area.

In Figure 2.32, when `restoreContext` is executed, a previously saved context that is stored in the kernel space area will be copied into SP and PC

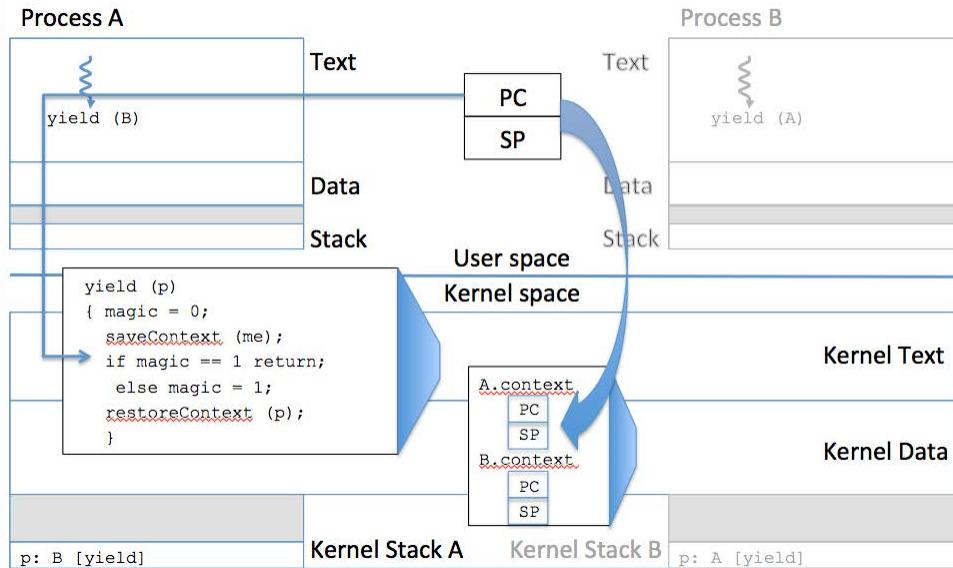


Figure 2.31: After saving the context of process A.

registers, as shown in Figure 2.33.

As shown in Figure 2.34, now that the context of process B has been restored, B will begin executing from where it left off. In effect, B just returned from its call to `yield`, which occurred in the past.

2.10 Summary

In this chapter, we learned that a process is the abstraction of a running program. We needed the concept of multiprogramming to allow multiple processes to be active, despite the fact that there's a single CPU. This was achieved by calling the `yield` procedure, which causes a process to give up the CPU to another process. This was achieved using the mechanism of

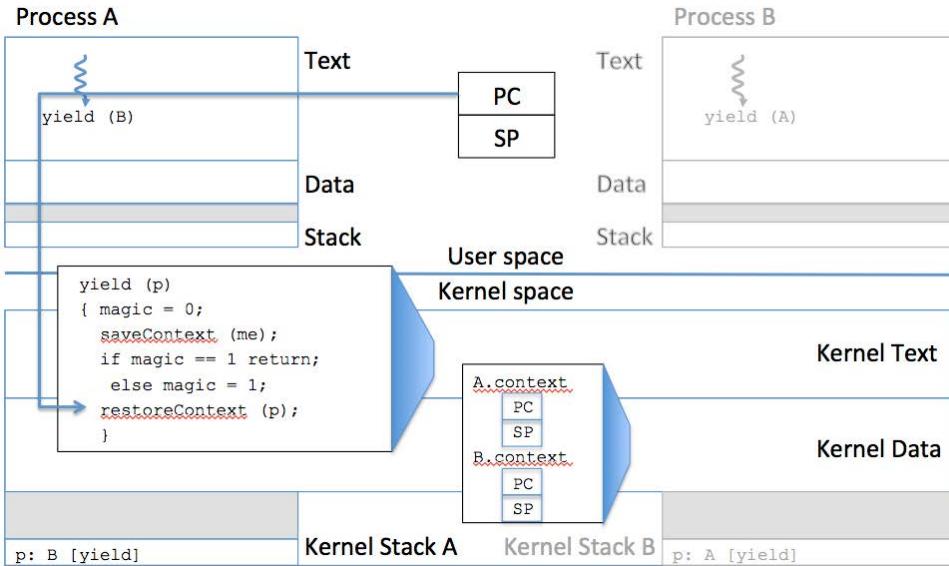


Figure 2.32: . About to restore the context of process B.

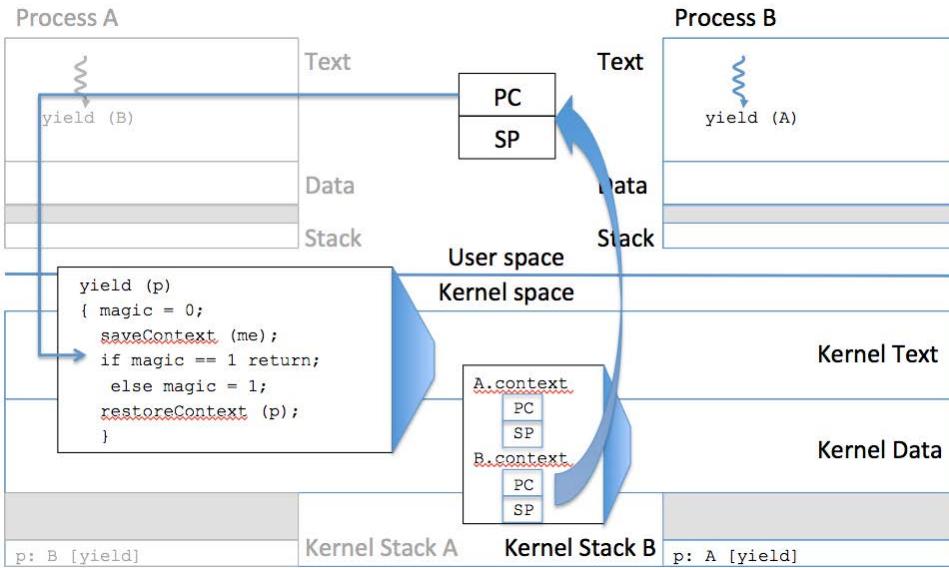


Figure 2.33: After restoring the context of process B.

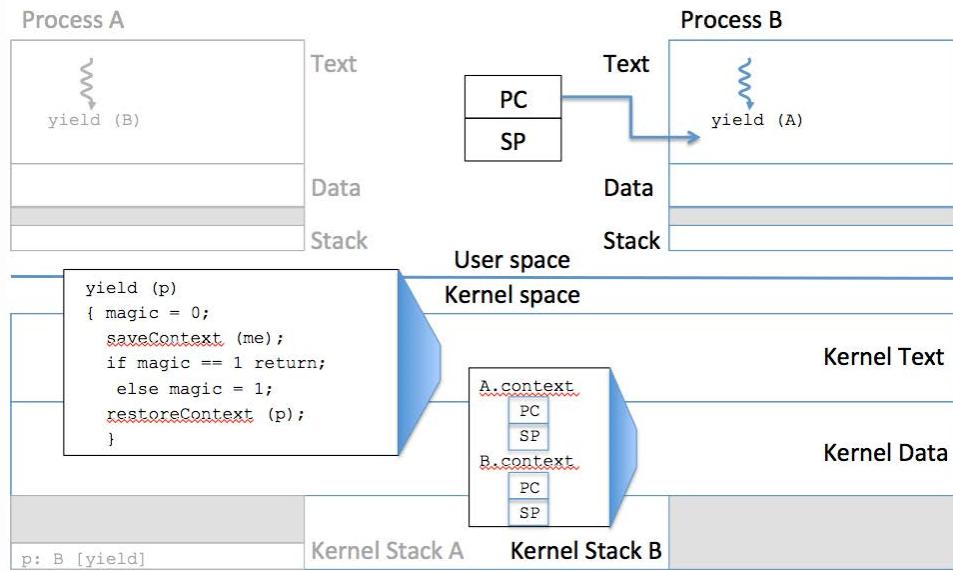


Figure 2.34: Return from `yield` in process B.

context switching, which transfers usage of the CPU from one process to another. It does this by saving the context of the currently executing process, and restoring the context of the next process to be resumed. Thus, we've encountered a new abstraction, that of a process, and a mechanism that allows us to realize this abstraction, that of context switching. However, we have not discussed any policy yet. That will have to wait for the chapter after next.

2.11 Exercises

1. What is the most basic functionality that the kernel provides to users?

2. Why is it difficult to run multiple programs on a machine with a single CPU and single (common) memory?**
3. What is a process?
4. How is a process different from a program?
5. What basic resources does a process need?
6. What is meant by the context of a process?
7. What makes up the CPU context?
8. What makes up the memory context?
9. Looking up the word “context” in a dictionary, how does this general definition relate to how the word is used in operating systems?***
10. What are the three sections that make up a process’s memory structure?
11. Why have three sections, as opposed to everything in one section?**
12. How are global variables different from local variables in a C program?*
13. In which section are global variables placed?
14. Can they be placed in the other sections?**
15. What is meant by the heap?*

16. Which section contains the heap?
17. Can it be placed in the other sections?**
18. What is an activation record?
19. Which section contains activation records?
20. Can they be placed in the other sections?**
21. Which sections can grow and shrink?
What event causes shrinking?**
22. For each section that can grow and shrink, what event causes growth?*
What event causes shrinking?**
23. What is contained in the SP register?
24. What functionality is enabled by the SP register?**
25. Why does the SP register need to be part of the hardware?***
26. What is contained in an activation record?
27. Justify the need for each item of information: why does it need to be contained in the activation record?**
28. Some machines provide an FP “frame pointer” register, in addition to an SP register: what is the purpose of the FP register, and why is it provided in addition to the SP register?***

29. What is meant by: Users would like to run multiple programs “simultaneously”?**
30. Why is “simultaneously” in quotes?***
31. Why is running multiple processes difficult on a machine with a single CPU?**
32. What is meant by multiprogramming?
33. If a process needs a resource and it is busy, why do we say the process “gives up” the CPU?**
34. What does the `yield` procedure do?
35. What is meant by context switching?
36. Why does yielding require context switching?*
37. What are the basic steps done in context switching?*
38. What must be the last instruction in context switching, and why?**
39. In simple (i.e., 2-process) context switching, why is it necessary to switch text, data, and stack sections?*
40. Which function is running during the switching of stacks?
41. Just *before* switching stacks, what activation record (i.e., which function call does it correspond to) is at the top of the stack?**

42. Just *after* switching stacks, what activation record (i.e., which function call does it correspond to) is at the top of the stack?***
43. What is the role of the “magic” variable in `yield` (i.e., why is it needed)?**
44. When running `yield`, where are the GP, SP, and PC saved?*
45. When running `yield`, from where are the GP, SP, and PC restored?*
46. How does `yield` know where to find the to-be-restored GP, SP, and PC?**
47. In the Example program in Figure 2.4, what does this program do?*
48. How many processes are involved in the example program?*
49. in Figure 2.5, why are the PC and SP pointing to the statements shown?
50. What is contained in the shared memory?
51. What is meant by shared memory?*
52. Is the shared memory part of Process A or Process B?**
53. Is “me” a global or local variable?
54. Is “magic” a global or local variable and why?*
55. Is a shared variable the same as a global variable?

56. in Figure 2.7, what causes the stack of process A to grow?*
57. Why are `saveContext` and `restoreContext` blocks of assembly language code and why?***
58. What does `saveContext` do?*
59. What does `restoreContext` do?*
60. Since `magic` is initialized to 0, why does it make sense to even check whether it equals 1?**
61. Will an optimizing compiler remove the if statement that checks whether `magic` equals 1, and why (or why not)?***
62. in Figure 2.13, what happened to the top activation record?**
63. in Figure 2.14, why is a new activation record created?**
64. What is meant by “(residual)” next to “`magic: 1`”?**
65. in Figure 2.14, what is the value of `magic` for process A?
66. How did it get set to this value?*
67. What does the TRAP instruction do?**
68. in Figure 2.28, the TRAP instruction is shown as part of Process A: does this mean that the programmer actually typed in “TRAP 20” as part of the program; why or why not?***

69. If the programmer did not type in “TRAP 20”, what generated that code?***
70. Why do we put the code for `yield` in the kernel?**
71. Why do we put process contexts in the kernel?*
72. Is the kernel a process, and why (or why not)?**
73. How many memory sections does the kernel have, and what is their purpose?***
74. In Figure 2.30, what is meant by “Kernel Stack A”?*
75. What is in Kernel Stack A, and why is it needed?**
76. Why is the stack for Process A empty despite having called `yield`, whereas in Figure 2.7 the stack for Process A grew after calling `yield`?**
77. Does it matter which stack is used to store the activation record for `yield`, why or why not?***
78. In Figure 2.34, what happened to Kernel Stack B?**
79. How does the behavior of `yield` differ between the example of Figures 2.5–2.27 and the example of Figures 2.28–2.34?**

80. Which method for implementing `yield` is better: that used in the example of Figures 2.5–2.27 or that used in the example of Figures 2.28–2.34?**
81. Which method do you think is used in a real operating system, and why?***



Chapter 3

Timesharing

In the last chapter, we saw how we needed the concept of multiprogramming to allow multiple processes to make use of a single CPU, by giving up the CPU when the process didn't need it. However, this depends on programmers adding `yield` calls to their code to allow the process to give up the CPU. In general, we cannot expect programmers to be so nice. In fact, there was a time when programs were written this way, and many programmers would cheat by not having their programs give up the CPU, so that they would get better performance. How do we solve this problem?

For this, we need the concept of *timesharing*. By timesharing, we mean multiplexing the use of the CPU over time in a regular way. Given multiple processes and a single CPU, each process gets a small portion of CPU time, thus allowing each process to seemingly make continual progress over time. In reality, each periodically gets a small fixed amount of time, and each

process progresses in small spurts.

3.1 Quantum

We called the small fixed amount of time a *quantum*. If the quantum is small enough, and we are able to rapidly switch the CPU amongst the various processes, the results will be the illusion of continuous parallel progress by all the processes over time.

In Figure 3.1, on the left side, we see three processes that are seemingly independent, running in parallel, and making progress over time. On the right side, we see that what is actually happening is that each process is getting to use the CPU, one at a time. We might imagine the CPU usage being divided like a pie, where the first process P_1 gets some portion, the second process P_2 gets another portion, and the third process P_3 gets the remaining portion, repeatedly. Looking at what happens over time, as shown in the graph on the lower right side of Figure 3.1, P_1 executes for a little bit, and then there's a context switch to P_2 , and P_2 executes for bit, and then there's a context switch to P_3 , and P_3 executes a bit, then back to P_1 , and so on. The small amount of time that each process executes is the quantum of time, and notice that it is a fixed amount.

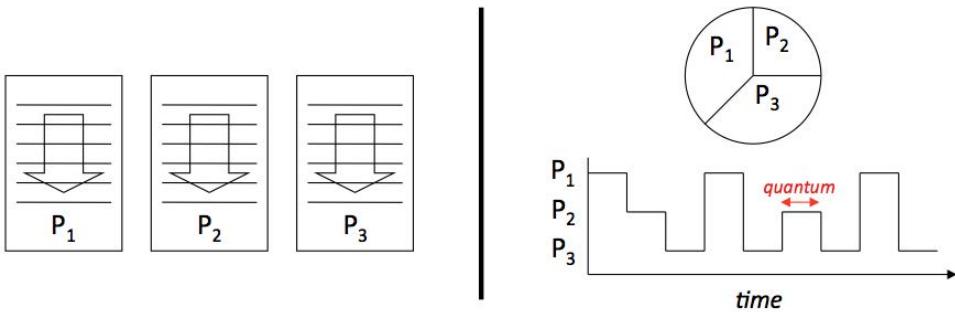


Figure 3.1: On the left, 3 processes making progress in parallel; on the right, each process getting some time on a CPU, one at a time.

3.2 Quantum Size

How big should the quantum be? We can reason this by looking at extremes.

If it is too big, such as, say, ten seconds, or even one second, the user will notice that each of their processes is making progress in spurts, breaking any illusion of continuous progress. This is not what we want. Rather we want it to seem that all the processes are each smoothly making progress, even if slowly, as time increases.

This argues for making the quantum small, but how small? What if we chose, say, one microsecond. Given such a small quantum, we as humans would not be able to resolve time intervals at this granularity, and so we would perceive continuous progress by all the processes. But, there is a problem! Each time the CPU is transferred from one process to another, time is required to perform the context switch. If, say, a context switch took one microsecond (which is a good ballpark figure on modern machines), then

for every microsecond of progress each process was able to make, there would be one microsecond of overhead to do the context switch, effectively, causing our computer to operate at half speed, or operating at 50% efficiency. This is clearly not good.

So, the answer lies somewhere in the middle. We want the quantum to be small enough so that we as humans cannot detect granularity in the progress being made by processes, but large enough that the overhead due to context switching is relatively low, as in 1/10 of one percent. Thus, a good number for the quantum is in the range of 1–10 milliseconds. If the context switching time is 1 microsecond, the overhead is at most 0.1%. Since humans are able to resolve times at about 0.1 seconds or 100 milliseconds, a quantum of even 10 milliseconds, at the high end of the range, is still a relatively small tenth of that. In fact, most operating systems use a quantum of 1–10 milliseconds, and as context-switching times are getting smaller, so does the quantum.

3.3 Process States

To implement timesharing, the kernel needs to keep track of the progress of each process. In fact, we need to know whether a process is actually able to make progress or not. For example, if the process is waiting for input, it makes no sense to give it the CPU; it is not able to make progress, no matter what, until it gets its input. Consequently, we characterize the state of a process as RUNNING, READY, or BLOCKED.

By the *RUNNING* state, we mean that the process is actually making progress because it has the CPU and is using it; it is *running*. The *READY* state means that the process is able to make progress, but currently does not have the CPU; while it is not running, it is *ready* to run. The *BLOCKED* state means that the process is *not able* to make progress, and so would not be able to use the CPU even if it were given it; it is *blocked* from running.

The kernel would use a process's state information as follows. When it is time to give another process a chance to run, say at the end of a quantum, the kernel would select from processes that are in the *READY* state (it would avoid processes in the *BLOCKED* state), and allow that chosen process to run. If, by the way, there were no ready processes, then the kernel would allow the currently running process, which is in the *RUNNING* state, to continue to run. How the kernel makes this choice is a scheduling *policy* decision, which we will discuss in the next chapter. Eventually, the kernel gets back control, i.e., it would get to use the CPU (after all, it can only run if it is actually allocated the CPU; as to who does this allocation, we will get to that soon), and at that point the kernel will select another process to run. And this cycle would repeat.

3.4 Process State Diagram

In Figure 3.2, we have what is called a *process state diagram*. This is one of the simplest possible process diagrams, with only three states, including the

basic states that we just described: RUNNING, READY, and BLOCKED. A more advanced analysis would have additional states, such as a STARTING state, an EXITING state, and perhaps other states that expand the BLOCKED state into sub-states that characterize the reason that a process is blocked, such as WAITING-FOR-DISK or WAITING-FOR-KEYBOARD. But for our purposes here, this basic state diagram in Figure 3.2 will suffice.

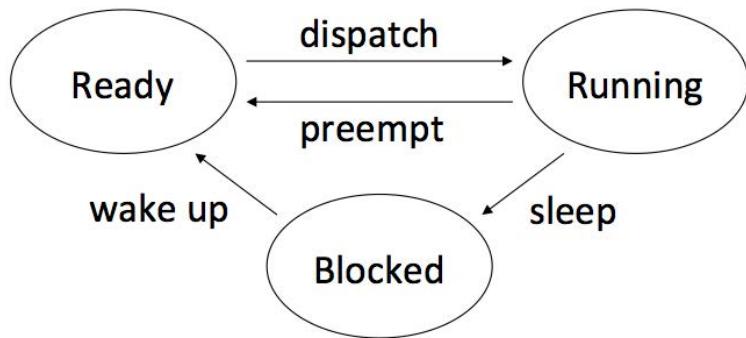


Figure 3.2: A process state diagram, showing states and transitions.

Let's look at each individual state transition. If a process is in the READY state, and the process is given the CPU, it moves from the READY state to the RUNNING state. This is called "dispatching the CPU to a process," and the transition is the *dispatch* transition.

If the process is in the RUNNING state, and at some point can no longer make any progress because it must wait for some resource, such as a disk to provide input, the process would give up the CPU and transition to the BLOCKED state. We call this the *sleep* transition. The process voluntarily gives up the CPU, because it cannot make any further use of the CPU.

Why do we say “voluntary?” If not voluntary, then what? Could the process be *forced* to give up the CPU? This would imply some other entity to be running to do the forcing. But, if there is only one CPU, and the process is using the CPU, nothing else could possibly be running. Since there is nothing to force the process to give up the CPU, this is why we say that the process “voluntarily” gives up the CPU.

Now consider a different situation. Let’s say that a process is running, and it is about to reach its limit as to how long it can run, by having used up an entire quantum. At that point, the CPU *must be forcibly taken away* from the process; we call this *preemption*. The *preempt* transition causes the process to go from the RUNNING state to the READY state. We will have to deal with how it is possible for the CPU to be forcibly taken away from the process, given our observation above that this requires some entity to be running to do the forcing, despite there being a single CPU. All in good time.

Finally, there is one transition left. If a process is in the BLOCKED state, it must be blocked for a reason. Perhaps it is waiting for a device to become available (the device may be working on another request). When the device becomes available to provide service to the process, there is no reason the process cannot begin running. Consequently, the process is moved from the BLOCKED state to the READY state. Notice that the process is not given the CPU immediately, as the READY state simply means that the process is eligible to run, but it still does not have the CPU. However, by being in the

READY state, the process is identified as being worthy of being allocated the CPU, which is useful for the kernel to know when it needs to make a decision as to which process should get the CPU next. When the kernel decides to give a process in the READY state the CPU, at that point the process would transition to the RUNNING state.

3.5 Logical vs. Physical Execution

An alternative view of the process state diagram is shown in Figure 3.3. Here, we show a matrix that describes logical versus physical execution. On one axis, we have *physical* execution, which means actually executing using the CPU. On the other axis, we have *logical* execution, which means the logical possibility for execution. By logical possibility, we mean: Is the process able or not able to execute? If a process is able to execute, but not actually executing (because it does not have the CPU), this corresponds to the READY state. If the process is not able to execute, then it certainly cannot actually be executing, and this corresponds to the BLOCKED state. If a process is able to execute and it is actually executing, this corresponds to the RUNNING state. Finally, if a process is not able to execute, it makes no sense to give it the CPU and so it could not actually be executing. This is why the upper right quadrant of the matrix does not correspond to any state, as no state would make sense.

		<u>Logical Execution</u>	
		Able to execute	Not able to execute
<u>Physical Execution</u>	Actually executing	Run	X
	Not actually executing	Ready	Blocked

Figure 3.3: Using logical vs. physical execution to determine process states.

3.6 Processes vs. the Kernel

Let's return to the issue of the difference between processes and the kernel. Recall that the kernel is code that supports processes, and runs as an extension of a process's execution. The kernel is not its own separate process. The way that a process enters the kernel is via *system calls*. These are procedure calls for procedures that are in the kernel, and that provide some system service. Some examples include the `fork` system call, which forks off, i.e., creates, another process; the `exit` system call, which is called by process when it is about to exit the system; the `read` system call, which is used by a process to obtain input from a file or an I/O device; the `write` system call, which is used by a process to submit output to a file or an I/O device.

In addition to procedures that are accessed via system calls, the kernel has its own internal procedures (called from within the kernel) that are used

for system management. We've already encountered one such procedure for the mechanism of context switching. In the next chapter, we'll consider another system management procedure for *scheduling*, which determines *which* process should get the CPU (and how long they may keep it), an example of policy rather than mechanism.

3.7 When Does the Kernel Run?

To be able to support processes, the kernel actually has to run. But, when does the kernel run? It runs whenever a system call is made, as we discussed. It also runs whenever a hardware interrupt occurs. Recall our discussion in Chapter 1, where we said that the kernel has two interfaces, an upper interface to allow interaction with processes, and a lower one to allow interaction with the underlying hardware. We have finally reached a situation where the lower interface is invoked.

Consequently, the kernel runs as part of the currently running process. If that process makes a system call, kernel space is entered and the kernel executes as an extension of the process. The process is still running, it's just that it's running in kernel space. If a process is running and an interrupt occurs, then the kernel gets control and the interrupt is handled. It is here that our convention that the kernel is not its own process is weak, as the cause of the hardware interrupt generally has no relationship to the currently running process. However, the currently running process is charged with

handling the interrupt, as some process must take responsibility for it.

If we were to adopt a different convention, where the kernel is its own process, then we would say that the hardware interrupt causes a context switch to the kernel process, and the kernel would handle the interrupt. That is a perfectly good way of interpreting the kernel, and has some desirable properties, but since most systems use the convention that the kernel runs as an extension of the currently running process, that's the one we use in this book. Another reason, perhaps more important, is that it is also closer to what physically happens in a machine.

To clarify this discussion, consider Figure 3.4. Here we have a process that was running in user space, and its execution is shown as the squiggly line in the text area. At some point, the process makes a system call, which will involve executing the TRAP instruction, or, a hardware interrupt occurs; either will cause control to transfer to kernel space, and the appropriate procedure is executed in the kernel.

3.8 Keeping Track of Processes

To allow the kernel to decide which process should run at the end of a quantum, it needs to keep track of all the processes that exist. Consequently, the kernel maintains a data structure that keeps track of this information. This is typically called the “process table,” as shown in Figure 3.5.

The process table might be implemented as an array of records, or a

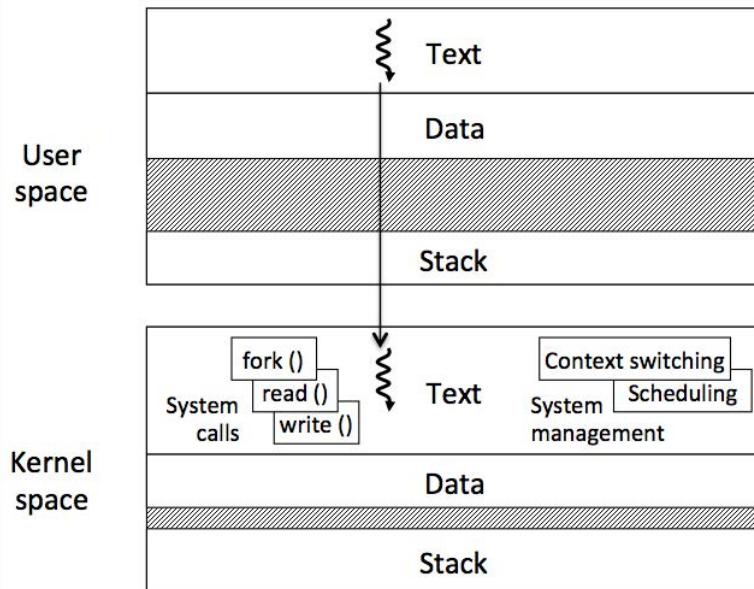


Figure 3.4: A process running in kernel space.

Process ID	State	Other info
1534	Ready	Saved context, ...
34	Running	Memory areas used, ...
487	Ready	Saved context, ...
9	Blocked	Condition to unblock, ...

Figure 3.5: An example of a process table.

linked list, or some other data structure. In this process table, the kernel records, for each process, its ID (i.e., identifier, which is a unique name so that all processes can be distinguished), its state, and other information. For example, saved contexts are saved in a field in this table. Other information includes pointers to the memory areas that the process is using. For processes that are in the BLOCKED state, it would include the reason why the process is blocked, or the condition that needs to be met to unblock the process.

3.9 How Does the Kernel Get Control?

We now elaborate on how the kernel gets control, i.e., how it gets the CPU and runs. One way this is done is for a process to make a system call. In that case, control goes to the kernel and a procedure corresponding to the system call executes.

It is possible that the system call cannot complete at that moment. For example, consider the `read` system call, which allows a process to obtain input from a file or an I/O device. If either the file (or to be more precise, the storage device containing the file), or the I/O device, is busy, the process must block. Note that the process is currently running in the kernel inside the `read` procedure. At this point, it cannot return from the kernel, because the `read` system call has not completed.

This is a good time for the kernel to allow another process to run. Given this, the currently running process must go into the BLOCKED state. Con-

sequently, the kernel would mark currently running process as BLOCKED, look in the process table to identify a process in the READY state, and then call `yield` to transfer control to that process. So, we see that `yield` is a procedure inside the kernel and that causes a context switch to another process.

3.10 Preemption

If preemption is to occur (because a process has used up an entire quantum of time), the kernel must somehow get control so that the CPU can be forcibly taken away from the currently running process. Recall that the kernel will run either because a process made a system call, or because a hardware interrupt occurred. We can use the latter to implement preemption.

Most modern computers have a hardware clock, i.e., a countdown timer, which can be set to interrupt after a certain amount of time. We can set the hardware clock with the quantum value, and when a process runs for an entire quantum, a hardware interrupt due to the hardware clock goes off, and the kernel is entered to handle the interrupt.

The kernel is able to determine that the reason it is running is that the clock interrupt went off, which means that the currently running process has used up its quantum, and so the kernel would mark the currently running process as transitioning to the READY state, and select a process in the READY state to run, causing a context switch to the chosen process. But

before the context switch occurs, the kernel will reset the hardware clock so that it will go off one quantum of time in the future. In this way, the kernel is guaranteed that it will run in at most one quantum of time.

This answers the question we raised above: How is it possible for the CPU to be forcibly taken away from the process, given that this requires some entity to be running to do the forcing, despite there being a single CPU? We now know that that entity is the kernel, and that since the kernel runs as an extension of the currently running process, it is effectively the process that forces itself to give up the CPU!

Consider another thorny question: What happens if a process makes a system call, which will occur before the quantum is over, and that process blocks such that another process is given the CPU to run? Is the new process given an entire quantum, and is the process that just gave up the CPU given any credit for not having used its entire quantum?

Typically, the answer is “no” in both cases. The reason is that, it would not help all that much to be so precise because a system call is a relatively rare event when one considers how many quantums go by before a system call is made. So, if every so often a process does not get a full quantum (which would be the case for both the process for which the CPU was taken away as well as the process that is given the CPU, which will now only run for the remainder of the existing quantum), it does not make much difference. It also reduces code complexity, as the kernel doesn’t have to keep track of partial quantums, and somehow crediting and debiting processes for the

partial amounts of time they use to achieve a higher level of fairness. Rather than having 100% fairness, perhaps we attain 99%, which is good enough.

3.11 User Mode vs. Kernel Mode

We can now look in closer detail as to how a context switch occurs. Regardless of whether the process makes a system call or a hardware interrupt occurs, in both cases, the kernel is entered. At this point, there's a certain amount of work that is done in hardware and some that is done in software. In hardware, there is a switch in the *protection mode* of the computer, from *user mode* to *kernel mode*. Let's elaborate on what this means.

The computer can be in one of two protection modes, user mode or kernel mode. In user mode, some machine instructions are not permitted, and so a limited portion of the instruction set is available. In addition, only a portion of the memory can be accessed. In kernel mode, there are no such restrictions: the entire instruction set is available, and all memory is accessible. This aspect of the hardware is used to allow the kernel to be protected. After all, we don't want programmers writing programs that can modify, or even read parts of, the kernel!

The kernel is placed in an area of memory that can only be accessed while in kernel mode. And when processes run, by default they run in user mode, which will limit the machine instructions that can be invoked. One such instruction is one that allows modification of the time entered in the

hardware clock, and we don't want processes to be able to modify their quantum time! Only the kernel should be allowed to do this.

When a process begins running its own code, the machine is in user mode. If the process makes a system call, the kernel is entered, and the machine mode is automatically set to kernel mode. This happens because the system call is not a normal procedure call. Rather, when the program is compiled, when the compiler translates (generates machine instructions for) a system call, it replaces what normally looks like a procedure call into a sequence of instructions that includes the TRAP instruction. The TRAP instruction causes control to go into the kernel, and importantly, causes a change from user to kernel mode. This change of mode from user to kernel mode is called an *amplification* of power, which is appropriate for the kernel. The only other way to enter the kernel is when a hardware interrupt occurs. An interrupt will also cause the protection mode to change to kernel mode. (If the mode was already kernel mode and an interrupt occurs, the mode remains kernel mode).

In both cases, either because of a system call or an interrupt, control goes to a fixed location in the kernel, the kernel determines what it should do based on the reason it is running (the kernel is able to determine whether it was entered because of a TRAP instruction or an interrupt) and then carries out the appropriate function. When the kernel returns, by having completed a system call or the work of the interrupt handler, it returns to what the process was doing at the point of the TRAP or interrupt, and resumes. The

mode is automatically set back to user mode so that the process does not obtain the same power as a kernel. (If an interrupt handler returns and the kernel was running before the interrupt occurred, the mode will remain kernel mode and the kernel will resume whatever it was doing prior to the interrupt.)

3.12 Context Switching in the Kernel

We can now resume our discussion of how a context switch occurs. We saw what happens in hardware, that the kernel gets control and that the protection mode is set to kernel mode. The kernel now runs, saves the context of the current process, and restores the context of the next process. It then returns, but where does it return? It returns to whatever the process was doing before the kernel was entered. But which process are we talking about? It is the process that was selected to run, the one being resumed, and the kernel knows what that process was doing before it entered the kernel at some point in the past, because its context had been saved as we saw in the last chapter. In this way, a context switch is caused.

3.13 Entering the Kernel via the TRAP Instruction

Let's look at an example. In Figure 3.6, we see a process that has just made the system call, `Getpid`. This is a system call that allows a process to learn of its process ID (identifier), something that the kernel knows and can provide to the process via this system call. We see that the `Getpid` system call is expanded into a sequence of machine language instructions generated by the compiler, that includes the `TRAP` instruction. In fact, `TRAP` takes a parameter, which in this example happens to be 20. This number corresponds to which system call is being invoked (i.e., all system calls are assigned a number, and the compiler includes this number when it generates the `TRAP` instruction).

The process enters the kernel because of the execution of the `TRAP` instruction. It is able to execute in kernel space because the `TRAP` instruction caused a modification of the protection mode to go from user to kernel. Because of the parameter 20, the kernel jumps to the `getpid` procedure inside the kernel (note the distinction between the `Getpid` system call and the `getpid` procedure call). The `getpid` procedure looks up the ID of the currently running process, which is stored in a global variable called `curproc`, and then executes the `IRET` instruction. `IRET` is similar to normal `RET` procedure return instruction, in that it finds what address to return to by looking in the top-most activation record on the stack, and jumps to that location after having popped off the activation record. But in the case of `IRET`, the protection

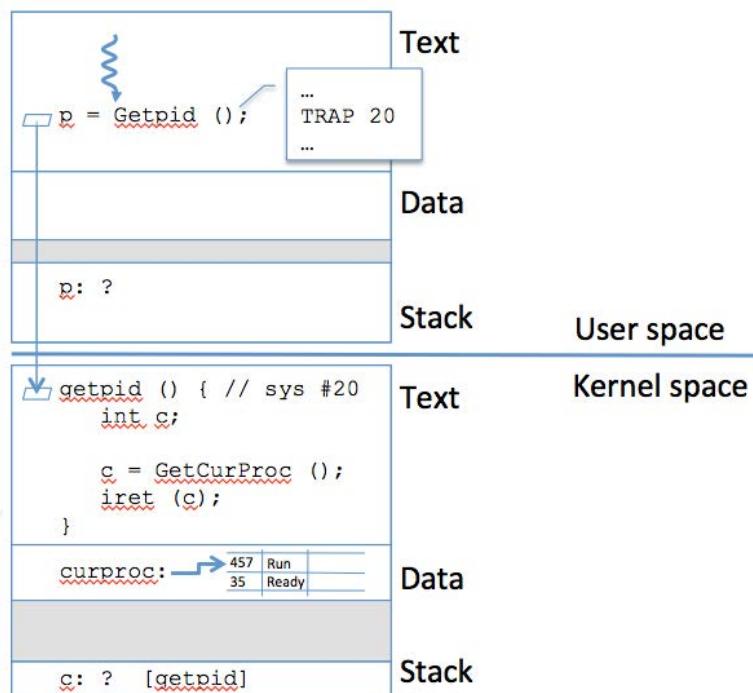


Figure 3.6: A process making a system call.

mode is also reset, back to user mode, because that's what it was prior to having entered the kernel.

The process can now continue executing in user space, now having learned of its process ID, after making a system call that syntactically looks just like having made a procedure call. This is only syntactic because what really happened was a jump into the kernel, execution within the kernel, and return from the kernel.

3.14 Entering the Kernel via a Hardware Interrupt

Now let's go through a similar example, except this time, rather than a system call having occurred, let's see what happens when a hardware interrupt occurs. In Figure 3.7, we see that a process was running in user space, and at some point, a clock interrupt occurs. This occurrence of the clock interrupt is asynchronous with respect to the execution of process instructions (i.e., it happens independent of the timing of the process instructions, not synchronized with any of them, including that the interrupt may happen somewhere in the midst of an instruction). At that point, the hardware suspends the current instruction being executed, and causes control to go into the kernel, similar to what happened with the TRAP instruction.

In addition to transferring control to the kernel, the hardware sets the protection mode to kernel mode, and transfers control to an interrupt handler

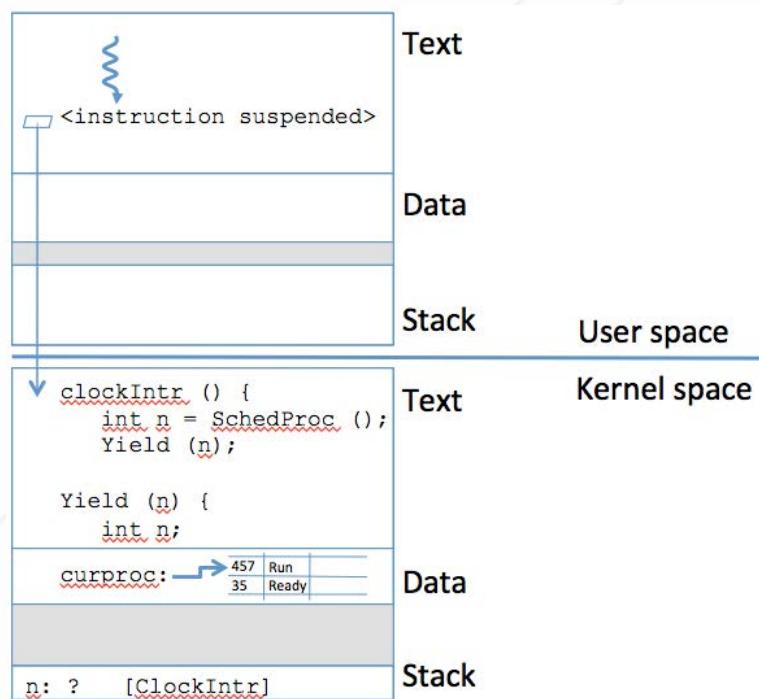


Figure 3.7: Entering the kernel because of a hardware interrupt.

corresponding to the interrupt that occurred, in this case the clock interrupt. Consequently, the clock interrupt handler executes, and the kernel determines that the reason the clock interrupt handler is running is that it must be the end of a quantum. And so the kernel decides to give the CPU to another process. It must first determine which process, and so it calls a procedure called `SchedProc`, which looks at the process table and determines which of the ready processes to run next. We will look at how this decision is made in the next chapter on scheduling. But for now, our only concern is that `SchedProc` returns the ID of the process that should run next. Now that the kernel knows this, it can carry out a context switch from the currently running process and to the selected process by calling `Yield`.

`Yield` will execute just as we saw in the previous chapter. `Yield` saves the context of the currently running process and then restores the context of the newly selected process. It also updates the process table to indicate that the current process should go from the RUNNING state to the READY state, and the new process should go from the READY state to the RUNNING state. `Yield` will then return, and using information in the activation record from the stack of the selected process, it returns to resume running from wherever that process had previously left off, as shown in Figure 3.8.

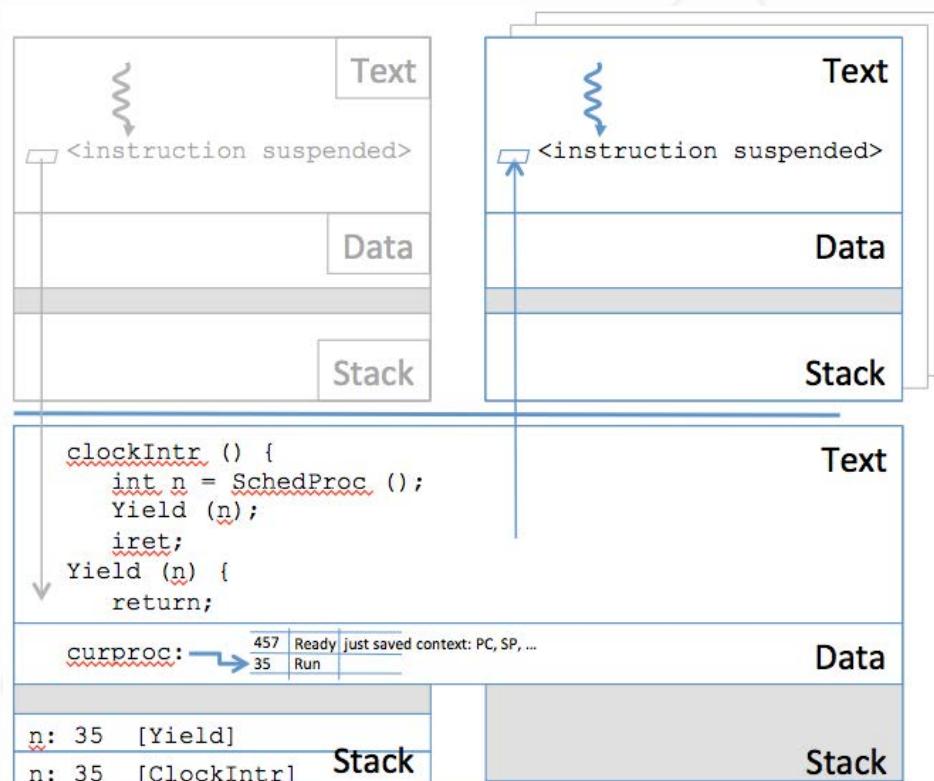


Figure 3.8: Context switch from one process to another after a clock interrupt.

3.15 The Big Picture

Figure 3.9 summarizes the big picture. Here we see that, in general, there will be many processes all but one of which are not using the CPU, and are in either the BLOCKED state or the READY state. Regardless of which state they are in, the last thing they all did before giving up the CPU was to execute code inside the kernel that caused a context switch to some other process. What they were doing, and where they should return to is recorded in their corresponding kernel stacks.

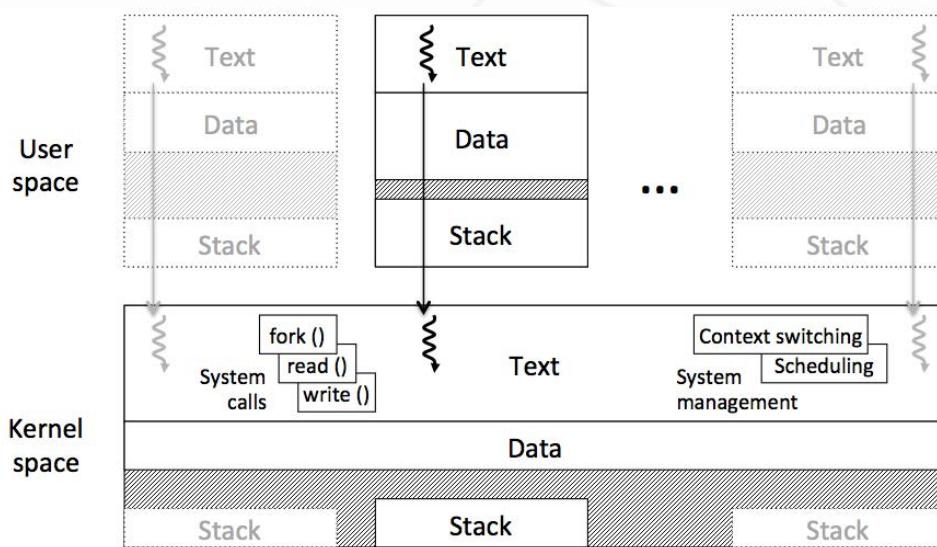


Figure 3.9: Multiple processes, all but one are in either the BLOCKED state or READY state, with one process in the RUNNING state running in the kernel.

The currently running process may either be in user space running user code, or kernel space running in kernel code; the example in Figure 3.9 shows

it running in the kernel. That process may return back to user space, or may give up the CPU to another process, one that is in the READY state, via a context switch.

3.16 Threads

We now consider the question of how to get parallelism given a single process. Recall our informal definition of process, which is that it is a program in execution. We can associate a program in execution with an execution path, the sequence of instructions that are executed as the process runs. So far, we've assumed that a process corresponds to a *single* path of execution in a memory composed of text, data and stack.

But what if we wanted *multiple* paths of execution, all within the single text area that contains the process's code, to exist in parallel? If we had multiple CPUs, this would be highly desirable, as the execution paths would actually run in parallel. The single data area would be available to the multiple execution paths, and so data could easily be communicated between those paths. However, we would need separate stacks, one per execution path, as each one would have its own independent record of call sequences that would generate a different stack of pending procedures. We call these execution paths, *threads*.

Before going further, let's consider the idea of attempting to obtain parallelism by having multiple processes. If we have multiple CPUs, we could

indeed assign a different CPU to each process and achieve true parallelism. So why do we need this concept of threads? The problem with multiple processes is that each process has its own memory, and barring an external shared memory area, the processes would either not be able to communicate, or would have to communicate by moving data between one memory and another. This would not be an issue with threads, as they already share a memory, the data area of a process, and can communicate by reading and writing that memory.

We can now be more precise. We define a *thread* as a single sequential path of execution. The abstraction of a thread is *independent of memory*. This is in contrast to the abstraction of a process, which includes both the idea of execution and memory. In fact, a thread is part of a process, the “execution” part. It lives in the memory of a process. By making the distinction between execution and memory, we now allow for the possibility that a process can have multiple threads. To the user, a thread is a unit of parallelism; the more threads, the more potential parallelism. To the kernel, a thread is a unit of schedulability; it is a thread that is assigned to a CPU.

Given our previous discussion on processes, we already know quite a bit about how we might implement threads. Just like we have system calls to create processes, like `fork`, and allow them to exit, like `exit`, we would have system calls that allow the creation of threads, such as `forkThread`, and allow them to exit, such as `exitThread`. And just like we have kernel support for the context switching and scheduling of processes, the kernel

would maintain management functions to support the context switching and scheduling of threads. Finally, just like each process requires its own user level and kernel level stacks, each thread now requires its own user level and kernel level stack. Given this, the kernel can schedule threads on separate CPUs if there are multiple of them.

This description is captured in Figure 3.10, which shows one process that has multiple threads, in this case three. The threads live in the memory of the process, and we know that a process's memory is divided into text, data and stack areas. Any of the process's threads can run any code in the text area and can access any data in the data area.

However, rather than the process having a single stack area, there will now have to be multiple stacks, one per thread. Similarly, in kernel space, the kernel has a text area for kernel code, a data area for kernel variables (that are not local variables for procedures, which are found in activation records), and then one stack per thread, as each of the threads may enter into the kernel at any point in time (by making system calls, or because an interrupt occurs).

In Figure 3.11, we generalize by showing multiple processes, each with one or more threads. Notice that given each process, for as many threads that that process has, it has that many user level stacks as well as kernel level stacks.

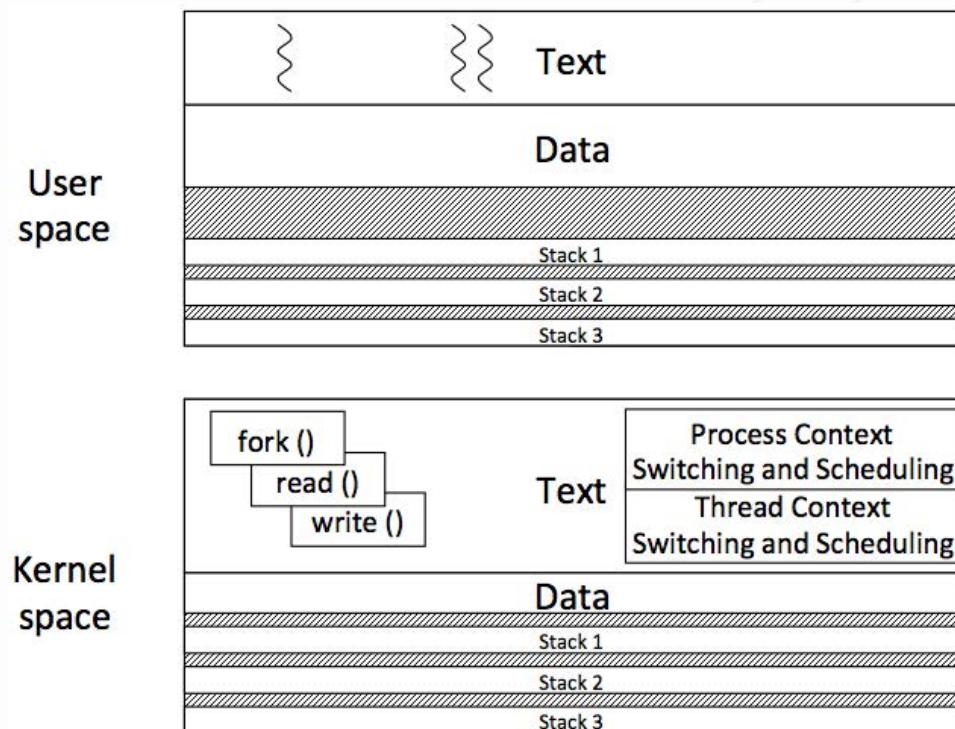


Figure 3.10: A single process with multiple threads.

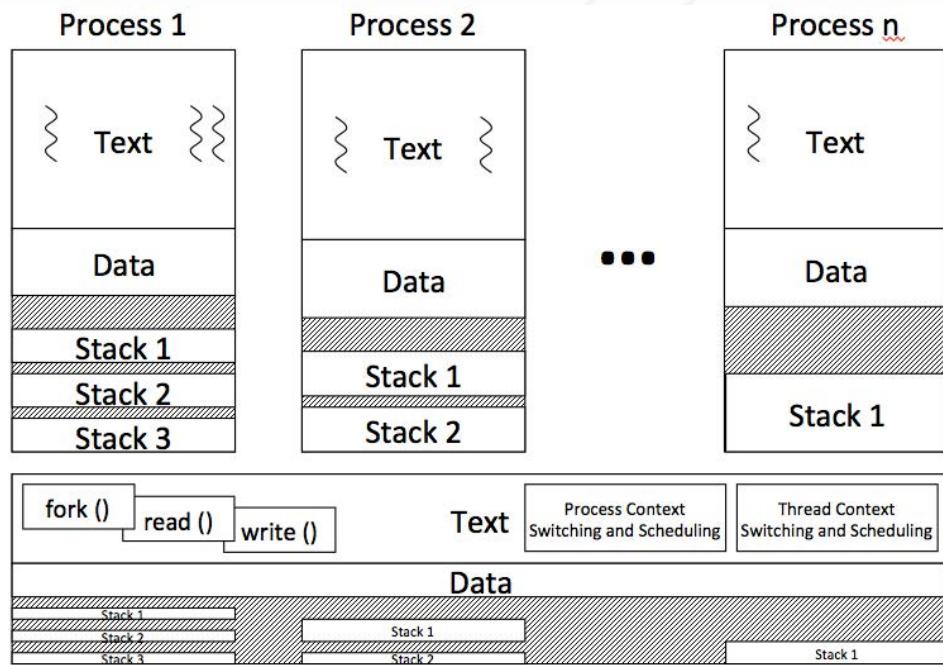


Figure 3.11: Multiple processes, each with one or more threads.

3.17 Kernel-level Threads vs. User-level Threads

There are actually two ways of implementing threads, either as part of a process, in which case the threads are called *user-level threads*; or as part of the kernel, in which case the threads are called *kernel-level threads*. Consider an operating system that supports processes but that does not support threads. We would still like to be able to support multi-threaded programs, i.e., programs that are written using threads, even though the operating system itself may not support threads. If a programmer believes that the best or most natural way to express parallelism in their program is to use threads, we do not want them to do otherwise because the operating system does not happen to support threads. Or, if a program was already written as multi-threaded, we would not want to force the programmer to rewrite their program, just because the operating system doesn't support threads.

We can solve this with user-level threads. In this case, support for threads would be provided via a user-level library, i.e., a package of code that, when compiled, would become part of the program and contain procedures that can be called by that program. These procedures would include support for threads, such as `forkThread`, `exitThread`, and even `yieldThread`, to allow one thread to yield to another. This would be done without any support by the kernel, and so despite that the operating system does not support threads, the abstraction of threads is still provided and available to the programmer.

The main downside of this approach is that, since the kernel does not

support the thread concept, and since it is the kernel that allocates the CPU, we cannot achieve true parallelism because the kernel cannot assign the CPU to a particular thread. The upside is that a program written with multiple threads is still able to execute, accepting the fact that the threads will have to share a single CPU.

Figure 3.12 shows how the system would be organized given user-level threads. The process would still be able to run multiple threads in the code in its text area, and its text area would include thread management code to support context switching and scheduling. Since the kernel in this case does not support the thread concept, the kernel provides the process with a single stack.

However, the user-level thread management code will divide this single stack into multiple smaller ones, per user-level thread. The kernel would still have a single text, data, and stack area, as the kernel only knows that there is a single process with presumably a single execution path. If one of the threads ever made a system call, that thread would have to complete its work in the kernel before another thread could be given the CPU. After all, the kernel would not be able to switch contexts between threads, because it is unaware of them. On the other hand, if the thread is running in user space, it can call `yieldThread` to cause a context switch between one thread and another.

The thread context switching would be occurring in user space, without having to jump into the kernel. In fact, this will be a very efficient context

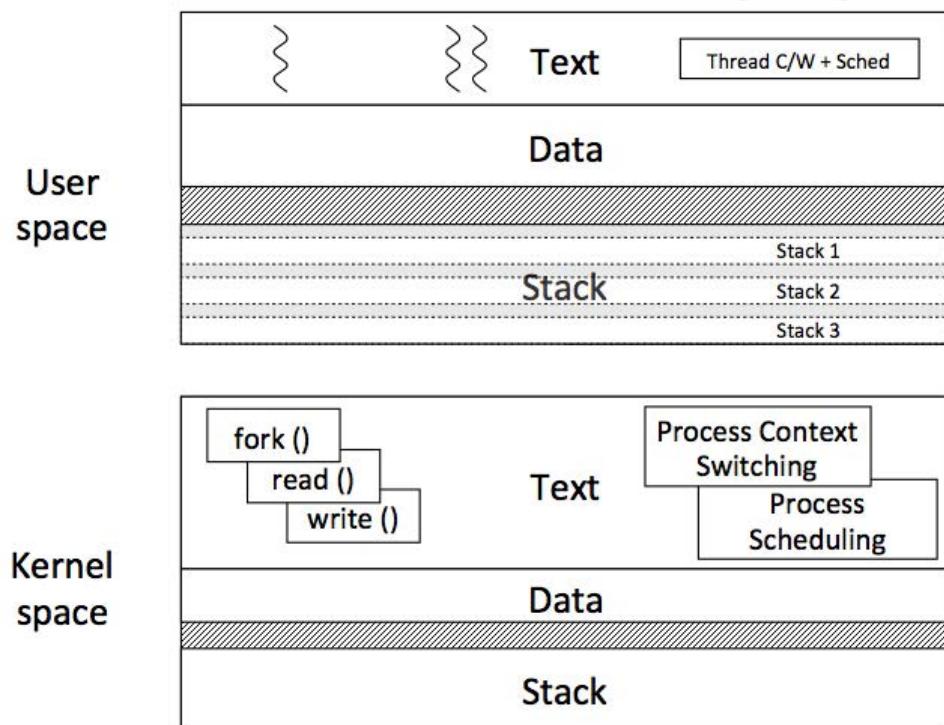


Figure 3.12: A single process with three user-level threads.

switch, because a system call is a much more expensive operation than a normal procedure call because it involves jumping into the kernel (during the call) and back (during the return). Crossing the user/kernel boundary involves a protection mode switch and then a change in the memory that is accessible, and these add to the cost in terms of time/overhead. These costs disappear if all the work is being done at user level. A context switch between user-level threads amounts to a procedure call, rather than a system call. And so, while we don't get the benefit of parallelism with the user-level threads, we do get the benefit of fast context switching.

In Figure 3.13, we generalize the diagram by showing multiple processes, each with multiple user-level threads. Notice that the user-level stacks are subdivided so that there is one per thread. However, the kernel-level stacks are per process, not per thread, because the kernel is not aware of the user-level threads. The kernel still carries out context switching and scheduling between processes, but the processes themselves do the context switching and scheduling of threads, which all happens in user space.

This leads to the question, which is better: user-level threads or kernel-level threads? Each has their pros and cons. An advantage of user-level threads is that, a program that uses them can run on any operating system, regardless of whether the operating system supports threads or not, as long as a user-level thread package is available. Another advantage is that user-level threads are very efficient, as the thread context switching occurs completely in user space. In the next chapter, when we discuss scheduling, we will see

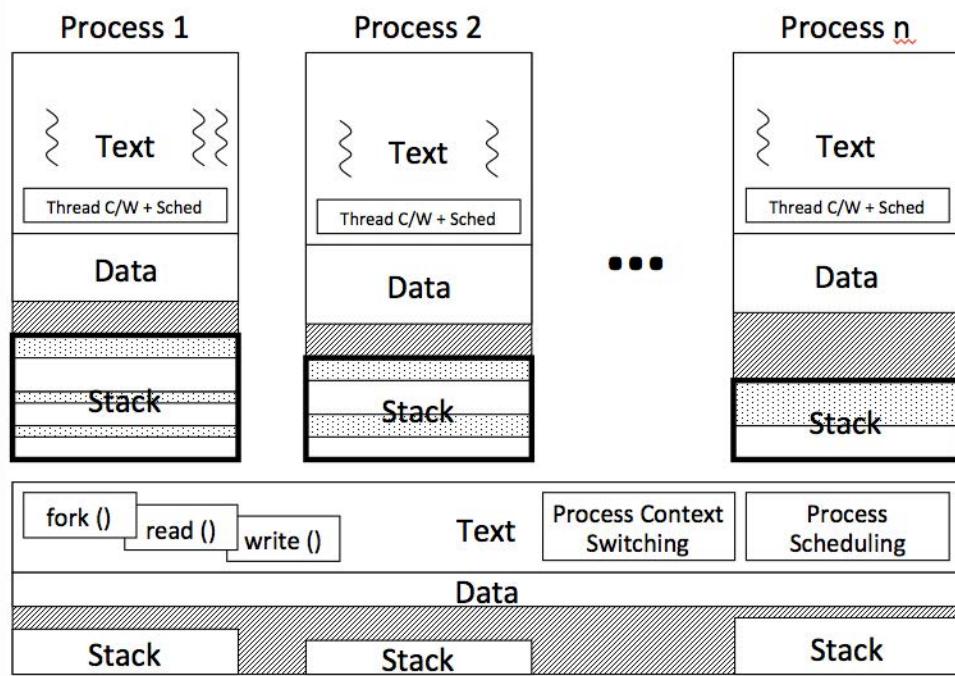


Figure 3.13: Multiple processes, each with one or more user-level threads.

that it is the kernel that dictates what the scheduling policy is. However, with user-level threads, the process itself could determine what the scheduling policy is for the threads within that process, giving the programmer the capability and flexibility of deciding what that scheduling policy should be. The big disadvantage of user-level threads is that we cannot obtain true parallelism, as the user-level threads cannot be assigned to different CPUs.

An advantage of kernel-level threads is that they do provide true parallelism, and so a group of threads that are part of the same process could be assigned different CPUs. However, a context switch between kernel-level threads is more expensive than context switching between user-level threads, because the context switching function requires jumps into and then out of the kernel.

Having said this, on modern computers, these costs are relatively small, and so it all comes down to whether the operating system supports kernel-level threads or not. If it does, the process will gain from the benefit of true parallelism amongst its threads. If it doesn't, then a programmer will have to use user-level threads, and benefit in that they don't have to modify their program to make it single threaded.

This distinction between user-level threads and kernel-level threads can be confusing. It is important to distinguish between thread support versus thread execution. Questions to ask include: Where is the thread abstraction being supported; is it supported by the kernel, or not? If supported by the kernel, we can use kernel-level threads; otherwise we can only use user-level

threads.

But independent of where the thread abstraction is being supported, there is the question: Where is the thread executing? A thread can execute in user space or in kernel space. Just because a thread executes in kernel space *does not mean it is a kernel-level thread*. It might be a user-level thread that made a system call, and is now running in the kernel. On the other hand, a kernel-level thread, i.e., a thread that is supported by the kernel, can run in user space. So, just because it is running in user space *does not mean it is a user-level thread*. Make sure you understand this important distinction!

3.18 Summary

In this chapter, we learned that time-sharing involves multiplexing the CPU amongst numerous processes by giving each process a small fixed amount of CPU time. We call this amount of time, the quantum. By making the quantum small so that the CPU is rapidly cycled amongst all the processes, we create the illusion of the continuous parallel progress of multiple processes.

We found it helpful to distinguish different states of a process, as either RUNNING, READY, or BLOCKED. This allows the kernel to limit itself to which process it gives the CPU, which should be only to ready processes. This is based on distinguishing between logical versus physical execution. If it is not logically possible for process to make progress, it is not worth giving that process the CPU.

We learned that the kernel can be viewed as an extension of any process, which runs when a process either makes a system call or when an interrupt occurs. We can implement preemption, the forcible taking away of the CPU from a process, by making use of a clock device interrupt, to limit how much time a process will receive, which we call the quantum.

Finally, we introduced the thread abstraction, where a thread corresponds to a single sequential path of execution. We made the distinction between kernel-level threads and user-level threads, and discussed their advantages and disadvantages.

3.19 Exercises

1. What is the difficulty in supporting multiple processes when there is a single CPU?**
2. What would it mean to say multiple processes are actually running simultaneously?***
3. What would it mean for it to seem to a human that multiple processes are running simultaneously despite there being one CPU?**
What might be a test for this?***
4. What is meant by “timesharing”?*
5. How is timesharing implemented?

6. What is meant to say “a process is: (a) RUNNING?”; (b) READY?”
(c) BLOCKED?”
7. How are the RUNNING and READY states similar?*
8. How do the RUNNING and READY states differ?*
9. How are the READY and BLOCKED states similar?*
10. How do the READY and BLOCKED states different?*
11. When the kernel decides which process to run next, does it look only at ready processes, or only blocked processes, or both, and why?*
12. What is a state transition?*
13. What is meant by the following state transitions: (a) Dispatch? (b) Preempt? (c) Sleep? (d) Wakeup?
14. What does the word “preemption” mean in general (its dictionary meaning)?
15. What does the word “preemption” mean in the context of operating systems?*
16. Why is there no transition from the READY state to the BLOCKED state?**
17. Why is there no transition from the BLOCKED state to the RUNNING state?**

18. What is the difference between “logical execution” and “physical execution”?*
19. Why does it not make sense to have a process that is physically executing but not logically executing?**
20. What are four examples of system calls?
21. What are two examples of system management functions?
22. Why are the functions for system management placed inside the kernel?*
23. Why are the functions for system calls placed inside the kernel?*
24. When does the kernel run?
25. Why does the kernel run only at these times?***
26. Is the kernel a process?
27. What is a justification for the view as to whether the kernel is a process?**
28. Can you offer a reason for why it might be good to adopt the alternative view?***
29. What is the difference between User Space and Kernel Space?*
30. When a process runs in User Space, what kind of code is it executing?*

31. When a process runs in Kernel Space, what kind of code is it executing?*
32. Why make a distinction between User Space and Kernel Space?**
33. How does a process go (change where it is executing) from User Space to Kernel Space?**
34. How does a process go (change where it is executing) from Kernel Space to User Space?***
35. What kind of information is found in the kernel's text area?* data area?* stack area?**
36. What is kept in the Process Table?
37. Can you justify why each item of information you mentioned is kept in the Process Table: Why is each needed, and what kernel decisions depend on it?**
38. What are the two ways that the kernel can get control, i.e., gets the CPU and starts running?
39. How does a process give up control voluntarily?
40. Can a process give up control without directly calling `yield`?*
41. How can the CPU be forcibly taken away from a process?* Who takes it away?**

42. What happens when a hardware interrupt occurs, as far as the kernel is concerned?*
43. How can the kernel ensure that it will eventually get to run?**
44. What are the steps, in both hardware and software, for doing a context switch?
45. What is meant by “user mode”?**
46. What is meant by “kernel mode”?**
47. If a process is running in user mode, what code is it allowed to execute?*
48. Can a process run in kernel mode, and explain why or why not?**
49. When switching from user to kernel mode, what is meant by “amplifies power”?***
50. Upon executing the TRAP instruction, where does the CPU start executing?**
51. If a hardware interrupt occurs, where does the CPU start executing?**
52. When the kernel selects the next process to run, what state must that process be in?**
53. What if there are no processes in those states, what process should run next?***

54. What does the RTI machine instruction do, and how is it different from a normal RET (return) machine instruction?***
55. What distinguishes a system call from a normal procedure call?**
56. What is the TRAP instruction do?*
57. How is it that the TRAP instruction appears in a user program (assuming the programmer does not insert it)?**
58. In Figure 3.6 (which also applies to the questions that follow), what is meant by “TRAP 20”?*
59. How is it that the procedure “getpid” is that one that starts being executed (why that procedure rather than some other one)?**
60. What is the purpose of curproc?**
61. Why is curproc in the Data section of the kernel (why can’t it be in the Stack section)?**
62. What data structure does curproc point into?*
63. What does it mean that curproc points to 457?**
64. Can you explain the contents of the kernel’s Stack area?**
65. In Figure 3.7 (which also applies to the questions that follow), what is meant by “instruction suspended”?**

66. What is `clockIntr`, and how is it that it starts executing?**
67. What is `SchedProc`?**
68. In Figure 3.8 (which also applies to the questions that follow), what is the result of calling `SchedProc`, and why?**
69. Could the result have been any different?***
70. Why is `Yield` called?**
71. Can you explain the changes in the data structure in the kernel's Data area?*
72. Why is the upper left side in shadow?**
73. Given your previous answer, why is kernel stack on the left side not in shadow?***
74. What caused the change in `curproc`?*
75. In Figure 3.9 (which also applies to the questions that follow), how many processes are there?
76. Why are there three stacks in the kernel?*
77. What is meant by “multiple paths of execution” within a single process?**
78. How is a single process with two paths of execution different from two processes that each have a single path of execution?***

79. Why does each path of execution require its own stack?***
80. What is a thread?
81. How is a thread different from a process?*
82. What is meant by a thread being a “unit of parallelism” to the user?***
83. What is meant by a thread being a “unit of schedulability” to the kernel?***
84. What is meant by “user-level threads”?*
85. What is meant by “kernel-level threads”?*
86. What is the primary difference between user-level threads and kernel-level threads?**
87. What is thread management?
88. In Figure 3.10 (which also applies to the questions that follow), how many processes are there?
89. How many threads are there?*
90. How many stacks are in User space, and why?**
91. How many stacks are in Kernel space, and why?**
92. In Figure 3.11 (which also applies to the questions that follow), how many processes are there?

93. How many threads does each process have?*
94. For each process, how many stacks do they have in user space, and why?**
95. For each process, how many stacks do they have in kernel space, and why?**
96. Why are the process CSSC (Context-Switching/Scheduling Code) and thread CSSC all in the kernel?***
97. In Figure 3.12 (which also applies to the questions that follow), how many processes are there?
98. How many threads are there?*
99. How many stacks are there in User space, and why?**
100. How many stacks are there in Kernel space, and why?**
101. In Figure 3.13 (which also applies to the questions that follow), how many processes are there?
102. How many threads does each process have?*
103. For each process, how many stacks do they have in user space, and why?**
104. For each process, how many stacks do they have in kernel space, and why?**

105. Why does each process have its own copy of the thread CSSC (Context-Switching/Scheduling Code)?**
106. Why is the difference between thread CSSC and process CSSC?***
107. Why is the process CSSC in the kernel?**
108. In a user-level thread system, where is the thread abstraction supported?**
109. In a kernel-level thread system, where is the thread abstraction supported?**
110. In a user-level thread system, can threads execute in: user space? kernel space? both?**
111. In a kernel-level thread system, can threads execute in: user space? kernel space? both?*
112. What are the advantages of user-level threads over kernel-level threads?*
113. What are the advantages of kernel-level threads over user-level threads?*



Chapter 4

Scheduling

We now consider the CPU scheduling problem. By *scheduling*, we mean which process gets the CPU, and when. Recall that we have assumed that there is only one CPU. Given multiple processes, which process should the CPU be assigned to, and how long should that process get to keep the CPU? There are a number of possibilities, or *policies*. A very simple policy is to allow a process to keep the CPU until it is done. Another is to give to each process a bit of CPU time and pass the CPU to the next, as we discussed in the previous chapter. Another is to give each process an amount of CPU time proportional to some external factor, such as how much a user is willing to pay. Which of these is best, and more generally, is there a best policy?

In fact, there is *no single best policy for all systems*. To determine the best policy for a particular system, it depends on what goals we want to achieve. The “best” scheduling policy for your personal computer will be

different than the best policy for a large timeshared computer, which will be different from the policy used for, say, a computer controlling a nuclear power plant. In fact, there may be multiple goals that need to be satisfied, and they might even be conflicting goals. Consequently, this is not an easy question to answer.

In this chapter, we will look at some of the most common policies for scheduling the CPU. In a real system, the policy used may be one of these, or will more likely be something a bit more complicated, including a hybrid of these. However, knowing these basic policies will give you a good background to understand and develop more complicated ones, and you will also have a good idea as to how to analyze how good a policy is.

4.1 Characterizing Scheduling Performance

We present the various policies by looking at a specific example of a workload. By *workload*, we mean a set of processes that require CPU time, and for each one, when they are created and how much CPU time they need. Then we will apply each policy to this workload, and see how the various policies compare to each other, as well as understand some of their general characteristics.

Let's first define some terms. By *arrival time*, we mean the time that a process is created. By *service time*, we mean how much CPU time a process needs in order to complete, at which point it can depart or exit the system. We define the *turnaround time* as the time between arrival and departure.

Consider a process that arrives, waits for the CPU to become available, and then uses it. It might use it in *bursts*, each of which is a continuous use of the CPU; after a burst the process may wait again (because another process must be given a chance to run), uses the CPU for another burst, waits, uses it again, waits, and so on, until it is done. At some point, the process departs after the CPU usage reaches the service time. The time between having arrived and having departed, which will include uses of the CPU and periods of waiting for the CPU, is the turnaround time.

We might then rate, i.e., characterize the performance of, a scheduling policy by seeing how well it *minimizes average turnaround time*. Why do we care about *minimizing* average turnaround time? Because, the longer it takes a process to complete its work, the worse we might consider its performance to be; we generally want processes to get in and out as quickly as possible. And we want a measure that accounts for all processes, and so that is why we use the average of the turnaround times, i.e., averaged over all the processes.

Consider the following workload, as shown in Figure 4.1. We are given three processes, A, B and C, all of which arrive at time 0. In other words, they are all created to appear in the system at the same time, at the beginning of time. The processes have different service times. Process A requires 5 units of the CPU time, process B requires 3 units, and process C requires 1 unit. We will not concern ourselves with what the time unit is. It might be 10 milliseconds, or it might be one second, or one minute, or one hour, etc. Regardless of the unit, ordering these processes to use the CPU in different

ways will result in different turnaround times, and changing the time unit will simply scale the turnaround times in the same way for all the processes; consequently, it doesn't matter what the time unit is (for the most part). The question will be, which type of ordering will result in the smallest turnaround time, expressed in generic time units?

Process	Arrival Time	Service Time
A	0	5
B	0	3
C	0	1

Figure 4.1: A workload with three processes.

Before going further, we've made an assumption that we know the service times of the processes in our workload. But, how realistic is this assumption? In general, it is not very realistic. When a process runs, we generally have no idea how much CPU time it will need. In fact, there is generally no way to predetermine how long a process will run. We may be able to use heuristics, such as the bigger the program is (in terms of number of lines) the longer it will run. This is a very weak heuristic, because some very short programs might have loops that cause them to run for a very long time. Or, we may keep a record how long it took to execute the program in the past, and use this to predict how long it might run the next time. While better, this is still a weak heuristic as the program's execution time may depend heavily on inputs that may change from run to run.

In general, there's no perfect way to anticipate how long a process will

run. But for the purposes of this chapter, we will assume we know the service times so that we can compare the various scheduling policies. In fact, we will see that some scheduling policies are adaptive, and will work well without knowing *a priori* what the service times are.

4.2 Longest First vs. Shortest First

We now return to our example. Let's consider two possible orderings, the first which orders processes by their service times from largest to smallest; the other which orders processes, again by their service times, but from smallest to largest. In addition, once a process is given the CPU, it uses it until it is done. This specification, which indicates the order that processes get the CPU, and for how long the CPU is kept, defines a *scheduling policy*. In this example, we presented the *Longest First* policy, and the *Shortest First* policy, respectively.

Figure 4.2 shows the results of Longest First and Shortest First. We see that for Longest First, process A executes for 5 time units, and since it is done because it required 5 units of CPU time, it exits the system and we record a turnaround time of 5, since it arrived at time 0 and departed at time 5. Next, process B gets the CPU, and uses it for 3 units. Since it arrived at time 0, waited for 5 time units for A to finish, then uses 3 time units, and so departed the system at time 8, we record its turnaround time as 8. Finally, process C gets the CPU, and uses it for 1 time unit, since its service time

is 1. Its turnaround time will be 9, because it entered the system at time 0 and departed at time 9.



Figure 4.2: Longest First vs. Shortest First. A green block means that the process is using the CPU during that time unit, while a yellow block means that the process is waiting for the CPU during that time unit.

Notice that the order in which the processes receive the CPU in Longest First, in terms of their service times. Since A has the longest service time, it went first, then B, and then C, because C had the least longest service time. We can then compute an average turnaround time, which is the average of the turnaround times of the processes. In this case, the average turnaround time for Longest First for this particular workload is $22/3$, or 7.3 time units.

Let's do the same for the Shortest First scheduling policy. Since process C requires the smallest amount of service time, it gets to go first. It enters the system at time 0, departs at a time 1, and so its turnaround time is 1.

Since process B has the next shortest service time of 3 time units, it runs next. Since it entered at time 0, and departs at time 4, its turnaround time is 4. And finally, since process A is the only process left, it gets the CPU and runs for its required 5 time units, and exits the system at time 9. Since it entered the system at time 0, its turnaround time is 9. We can now compute an average turnaround time for Shortest First, which is $14/3$, or about 4.7 time units.

And so we have our answer: For this workload, Shortest First does better than Longest First, where we defined better as having the shorter average turnaround time. We have to be careful with interpreting this result. Does it say that Shortest First will always be better? Not necessarily. It only says, *for this particular workload*, Shortest First is better. And we must keep in mind that by better, we mean something very specific, and that is that shorter turnaround times are preferred over longer ones. If our metric for better was longer average turnaround times, then Longest First would be better, again for this workload.

Given all these caveats, can we say we learned anything that can be generalized from this example? Can we say that Shortest First will always be better than Longest First, regardless of the workload? That would be a powerful result, but one that we cannot deduce by looking at a single example. However, the examples that we've chosen in presenting the various scheduling policies were carefully constructed, one might say contrived, such that they are representative of the behavior of a broad set of workloads,

beyond that of the example. If you want to check whether they do indeed generalize, you must do the work to figure this out (left to you as an exercise!). But, rest assured that they do indicate representative behavior.

How might you do this? Let's take the example we just discussed. From this example, we are led to believe that Shortest First is generally better than Longest First. But rather than simply taking our word for it, that somehow this example was carefully chosen to be representative of general workloads, can we be more definitive? This is a rare instance where the answer is *yes*.

4.3 Optimality of Shortest First Scheduling

In fact, *Shortest First is provably optimal*. Given n processes with service times S_1, \dots, S_n (processes are numbered 1, 2, 3, ..., n), average turnaround time, T , is computed as follows:

$$T = (S_1 + (S_1 + S_2) + (S_1 + S_2 + S_3) + \dots + (S_1 + \dots + S_n))/n \quad (4.1)$$

We can rearrange the terms, extracting all the S_1 's, then S_2 's, etc., to arrive at the following:

$$T = ((nS_1) + ((n - 1)S_2) + ((n - 2)S_3) + \dots + (S_n))/n \quad (4.2)$$

Since S_1 has maximum weight (it is multiplied by n , all the others are

multiplied by values less than n), minimize that factor (weight times service time) by having it correspond to that of the shortest process. Next, since S_2 has next-highest weight ($n-1$), minimize it (after S_1) by having it correspond to the service time of the second-shortest process. Do this for the rest, and we arrive at the following conclusion: to minimize the average turnaround time, order processes from shortest to largest.

And so we have proven that, in general, it is better to order processes in shortest first order according to their service times. It turns out that this is one of those rare situations that we can actually prove generality. Usually, scheduling policies do not lend themselves to such neat mathematical modeling, and so a proof, if indeed there even is one, is often difficult to derive, unlike this case.

Before considering other scheduling policies, let's make the workload example a little more interesting. Figure 4.3 shows a slightly modified workload, where the 3 processes arrive at *different* times in the system. We now have process A arriving at time 0, process B arriving at time 1, and process C arriving at time 2. We'll keep the same service times for each process as we had before. This is the workload that we will use in our upcoming examples.

If the processes are scheduled according to their arrival time, then process A would run first since it arrived before all the others, then process B would run since it came next, and then process C would run. Notice that B will have to wait until A is done, and then it will run. And C will have to wait for both A and B, and then it will run.

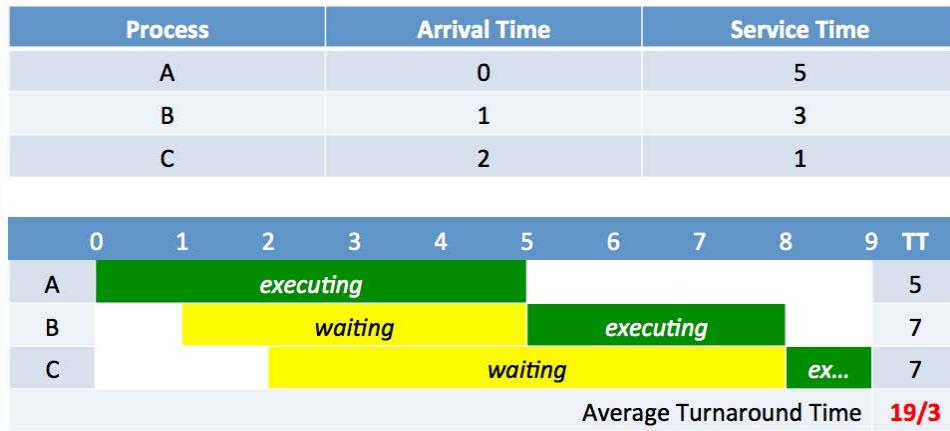


Figure 4.3: A workload with processes arriving at different times.

We can still compute turnaround times for each process, but this time taking into account that their arrivals are at different points in time. Process A's turnaround time is 5 because it arrived at time 0 and departed at time 5. Process B's turnaround time is 7, because it arrived at time 1 and departed at time 8, and so $8 - 1 = 7$. Finally, process C's turnaround time is 7 because it arrived at time 2 and departed at time 9. We can then compute an average turnaround time of $19/3$, or 6.3 time units.

4.4 First Come First Served

Figure 4.4 shows the *First Come First Served* (FCFS) policy. This is a very simple and very common policy. It allocates the CPU to processes in their order of arrival, and allows processes to keep the CPU until they are done. In this example, the average turnaround time is 6.3. Is this good or bad?

Well, compared to what? Only when we see other policies and their average turnaround times (and under the assumption that the workload generates representative results), can we say whether this average turnaround time is good or bad.

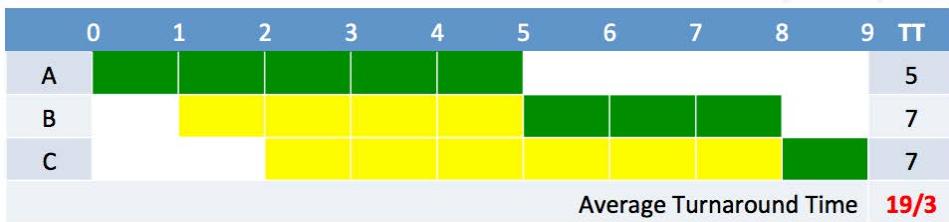


Figure 4.4: Schedule using First Come First Served (FCFS).

However, we can observe some characteristics about the scheduling algorithm itself. First of all, it is very simple. Simple is good, because it implies a simple implementation, and perhaps a simple analysis of its behavior (not always true, but often is). For FCFS, we can imagine a very simple implementation. We just maintain a first-in-first-out queue, adding a process at the tail when it enters the system and removing the process at the head when it exits the system, and giving the CPU to whichever process is now at the head of the queue.

FCFS is an example of a non-preemptive scheduling policy. By *non-preemptive*, we mean that the CPU will not be forcibly taken away from a process; once a process has the CPU, only it can give it up, and in this case, the process gives up the CPU when it completes. This is good to know, because if preemption were possible, we would require a mechanism that

allows the kernel to take the CPU away from a process. We encountered such a hardware mechanism in the previous chapter, namely the clock interrupt. If we have a computer that has no clock interrupt, then we would have to limit ourselves to non-preemptive scheduling policies.

FCFS also has the property that there can be no starvation. By *starvation*, we mean that a process may never get to use the CPU. However, for FCFS, a process will eventually make its way to the head of the queue, regardless of which processes come after (or before) it, and so will eventually get to use the CPU. It may have to wait a long time, but it will not wait forever (assuming of course, finite service times, which we assume throughout). If it is possible for a process to have to wait forever, then we say that the algorithm does not prevent starvation.

We can also say that FCFS behaves poorly for short processes, i.e., processes that have short service times. This is because, if a process finds itself behind another process that requires a long service time, it will have to wait all that time, despite that it only needs a small amount of CPU time to complete and exit. You may have experienced this situation at the supermarket. If all you want to buy is a carton of milk, but when checking out, you get stuck behind someone with a cart of, say, a hundred items, and unless they happen to be nice and let you pass over them, you'll have to wait, despite that you only have one item.

We will see that there are scheduling policies that are better suited for shorter processes. And we know, from our first example, that favoring shorter

processes is optimal in reducing average turnaround time. Unfortunately, FCFS does not have this property. But it is simple and non-preemptive, and so we want to remember it for these reasons.

4.5 Round Robin

Figure 4.5 shows the *Round Robin* (RR) policy. In this policy, each process gets a bit of CPU time in turn until done. Specifically, time is broken up into times slices or quantums, and each process gets one quantum of CPU time usage. The CPU is then given to the next process, which uses up to one quantum of CPU time, then going to the next process in similar fashion, until every process has gotten a turn at using the CPU. The cycle is then repeated.

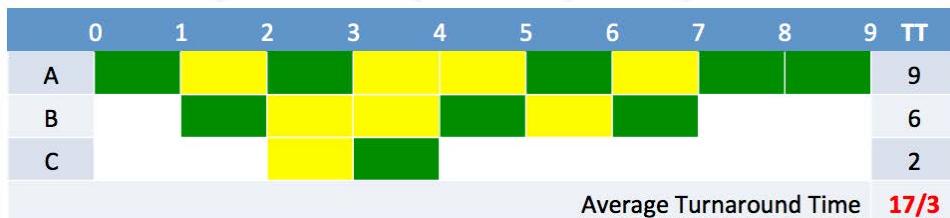


Figure 4.5: Schedule using Round Robin (RR).

In Figure 4.5, process A gets the CPU first because it is the only one present at time 0, and A runs for one quantum, at which point process B arrives. Process B then gets the CPU for one quantum, and at time 2 the CPU is given back to A because, when C arrives, it is queued behind the

other queued process, namely A. After A uses the CPU for one quantum, process C is given CPU for one quantum. Since process C only needs the CPU for one quantum, it exits the system at time 4. The CPU is then shared by process B and A, each getting one quantum of CPU time repeatedly, until they complete. At time 7, process B is done, and so process A uses the CPU exclusively until it completes at time 9. The actual order that establishes the cycle amongst all the processes is not that important; what matters most is that every process gets one turn before any process gets a second, at which point the cycle repeats.

Based on this workload, the average turnaround time for RR is 5.7. We can compare this with 6.3 for FCFS, and see that, for this particular case, RR is better. And indeed, RR is generally better at reducing average turnaround time than FCFS. The intuition for this is that, in RR, no single process can monopolize the CPU for a long period of time. The average turnaround time will go up when a single process that is very long runs while all the other processes have to wait. While this can happen with FCFS, it cannot happen with RR. Shorter processes will generally get in and out more quickly with RR. In fact, if there are n processes, a very short process, one that requires a single quantum, will have to wait at most $n - 1$ quantums before it gets the CPU, and then will immediately exit. No such claim can be made for FCFS.

The main reason this is possible is that RR is a *preemptive* scheduling policy. At the end of each quantum, the CPU is taken away from the current process, and so no process can keep the CPU without allowing others to

run. Given that it is preemptive, we do require a mechanism to support preemption, such as the clock interrupt, to implement RR.

RR is fairly simple to implement (though not as simple as FCFS). All that we need is a circular list; at each quantum, select the next process in the list. RR is not susceptible to starvation, because every process is guaranteed to make some progress within every cycle of n quantums, given n processes.

4.6 Shortest Process Next

Figure 4.6 shows the *Shortest Process Next* (SPN) policy. In this policy, the next process to run is the one with the shortest service time. This is a non-preemptive policy, and so once a process gets the CPU, it keeps it until it completes. At that point, of the existing processes, the one with the shortest service time is selected.

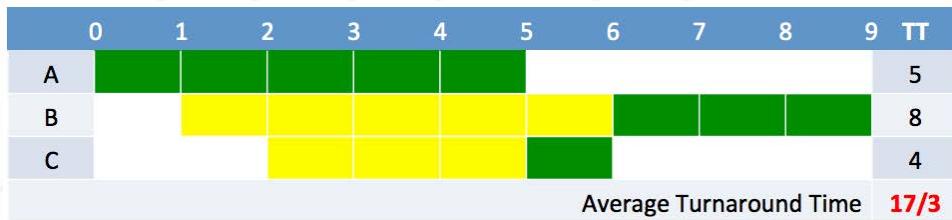


Figure 4.6: Schedule using Shortest Process Next (SPN).

In our example for SPN, the average turnaround time is 5.7. This is better than what we saw for FCFS, and equal to what we saw for RR. However, we should not compare this policy to RR, because preemptive and non-

preemptive policies are not fairly comparable. However, we can say that SPN will generally do better, or at least will never do worse, than FCFS. In fact, we can make this claim very confidently, because we know that selecting the shortest process first is an optimal strategy, precisely what SPN does. Consequently, FCFS can never do better than SPN. Of course, it might equal SPN for some particular workload, such as one where processes just happened to arrive in the order of their service times, from shortest to longest. Because of this, while SPN *generally* is better than FCFS, we cannot say it is *always* better.

SPN is an example of a policy that cannot be implemented unless service times are known before hand. Given that this is generally not possible, we study SPN mainly because we can use it as a benchmark compared to other non-preemptive algorithms. If we find another algorithm that comes close to SPN in performance, but does not require knowledge of service times, then we know that we have something good.

Notice that starvation is possible with SPN. A very long process may never get to run if there are always new arrivals of processes with shorter service times that arrive *before* the long process starts (as once it starts, it will not be preempted despite the arrivals of shorter processes).

4.7 Shortest Remaining Time

Figure 4.7 shows the preemptive version of SPN, called *Shortest Remaining Time* (SRT). With this algorithm, a scheduling decision is made at the end of each quantum, just like RR, and the process with the shortest remaining service time is selected for the next quantum. Notice that SRT is a preemptive algorithm.

	0	1	2	3	4	5	6	7	8	9	TT
A											9
B											4
C											1
Average Turnaround Time											14/3

Figure 4.7: Schedule using Shortest Remaining Time (SRT).

In our example for SRT, the average turnaround time is 4.7. This is better than any average turnaround time we've seen so far. In fact, SRT is optimal in minimizing average turnaround time. Just like SPN, we would use SRT as a benchmark compared to other policies, but in this case, to preemptive ones. Consequently, we know that RR, another preemptive policy, can only do worse or be equal to SRT, but never better. If we devise a new preemptive algorithm, and its average turnaround time is, say, within 5% of that of SRT, for a wide variety of workloads, we would consider the new policy a very good one, because it seems to behave close to the optimal. We also note that, just like SPN, SRT suffers from starvation.

At this point, one might wonder about the extra overhead that is involved in preemptive algorithms, where there will generally be a lot more context switching. After all, context switching does cost a bit in time. In our analyses, we are ignoring this overhead. But more generally, context-switching overhead is so small that it is not unreasonable to ignore this overhead, as the other properties of the scheduling policy will generally be the major determinants as to how it performs.

4.8 Multi-Level Feedback Queues

Figure 4.8 shows the queue structure for the *Multi-Level Feedback Queues* (MLFQ) scheduling policy. This one is more complicated than the others that we've seen so far. In MLFQ, we have n priority queues, where increasing queue numbers are decreasing in priority; 0 is considered the highest priority, and $n - 1$ is the lowest priority. When a new process arrives, it goes to the highest priority queue, namely queue 0. When a scheduling decision is made, the kernel identifies the highest priority queue that is not empty, and removes the process at the head of that queue, allowing it to run. If that nonempty queue is numbered k , the process will run for 2^k quantums.

Consequently, a process that was in queue 0 will run for one quantum. Assuming it didn't complete and exit, it must now go back to a queue, and so it will be placed in queue 1. More generally, if a process came from queue k , it will run for 2^k quantums, and assuming it didn't complete, it will be

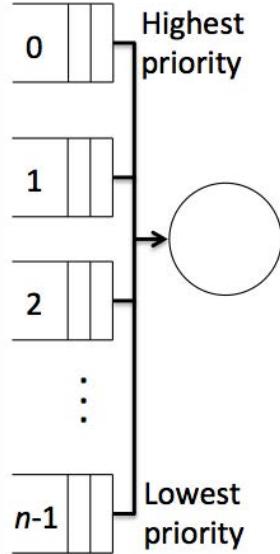


Figure 4.8: Multi-Level Feedback Queues (MLFQ).

placed at the end of queue $k + 1$, until it reaches the lowest priority queue, $n-1$, after which it can go no lower and so will just go back to the same queue.

If the process is allocated a certain number of quanta, say 2^k , and does not use all of them – for example, say it blocks due to I/O – when that process becomes ready again, there are different variations for treating that process as to which queue it is placed. It might get placed in the same queue that it came from, in this case k , or it might be promoted to a queue of higher priority than k , such as $k-1$, or even the highest priority, 0. This, in a certain sense, is to credit a process that did not use all of its allocated CPU time.

From this, we can see that MLFQ is an *adaptive* policy, seeking to order the processes according to their service times. Processes with short service times will tend to occupy the higher priority queues, and those with longer service times will find their way to lower priority queues. MLFQ seeks to approximate Shortest First, which we know is optimal, without having to know the service times before hand, effectively learning them as time goes on.

Figure 4.9 shows how our workload behaves when using the MLFQ scheduling policy. Notice that after each process has gotten to run for one quantum, process C will have exited, and the others, A and B, will then run for *two* quantums, because they were demoted from queue 0 to queue 1. Notice that MLFQ is a preemptive algorithm (one can also imagine a non-preemptive version of MLFQ, but here we assume it is preemptive). The result in this example is an average turnaround time of 5.3 quantums. Comparing MLFQ to our other preemptive policies, it does better than RR, but not as good as SRT, which is what we would expect. It does better than RR because MLFQ is able to favor processes with shorter service times in a stronger way than RR is able to.

MLFQ is a fairly complex algorithm, more complicated to implement than RR. It is a highly responsive algorithm, in the sense that short processes get fast attention. It favors shorter processes over longer ones, so it approximates SRT, making it highly desirable. However, it is susceptible to starvation. This can be addressed by artificially boosting the priority of any process

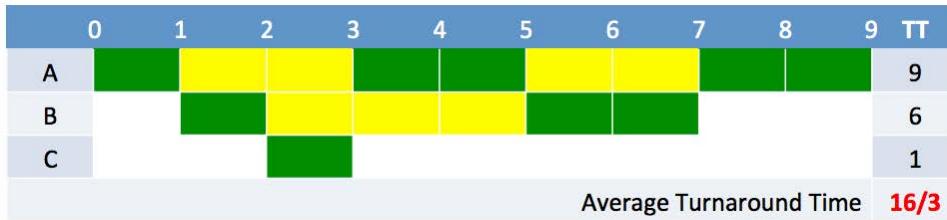


Figure 4.9: Schedule using Multi-Level Feedback Queues (MLFQ).

that has not gotten a chance to run in a very long time. Most modern scheduling algorithms are based on some form of MLFQ.

Up to this point, we've defined "better" as having a smaller average turnaround time. However, recall that we can have different goals, and that is why we can never say in absolute terms that a policy is best, as it depends on the goal that we are trying to achieve. Imagine that we had a goal based on some external criteria. For example, say we wanted to prioritize processes based on a user's willingness to pay for service. How would we develop a scheduling discipline to take this into account?

4.9 Priority Scheduling

Figure 4.10 shows the *Priority Scheduling* (PS) discipline. PS is very simple: the process with the highest priority is always selected to run next. In our example of three processes, let's assign priorities as follows: process A is assigned medium priority, process B is assigned high priority, and process C is assigned low priority. When process A arrives at time 0, it runs (if only

because there is no other process to compete with it). At time 1, process B arrives, and since it has a higher priority than process A, A is preempted and B is allowed to run next. At time 2, process C arrives, but it does not preempt B because it does not have higher priority. In fact, it has the lowest priority, so it will not run until all other processes have completed. When process B completes at time 4, there are two processes in the system: process A, which has medium priority, and process C, which has low priority. Consequently, process A is chosen, it will run until it completes, and then process C executes until it completes.

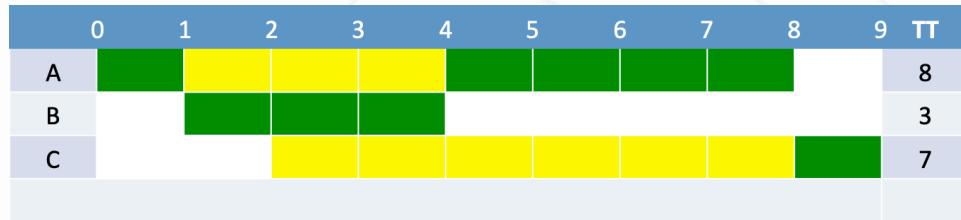


Figure 4.10: Schedule using Priority Scheduling (PS).

We could compute an average turnaround time, but it is irrelevant to do so, as the goal here is not to minimize average turnaround time, but to satisfy the given priority requirements of the processes. How are the priorities assigned? They are based on some external criteria, which were unspecified in our example: we simply declared that process B had the highest priority, process A had medium priority, and process C had lowest priority. Perhaps the users that owned those processes had paid for the given priorities, where a request for higher priority demands a higher price. In summary, whenever

we have external criteria that define the goal for scheduling, we would use priority scheduling to achieve it.

4.10 Fair Share

We now consider another way of assigning which process gets the CPU when, which is called *Fair Share* (FS), also called *Proportional Share*. The goal here is to give each process a share of the CPU based on given or requested proportions. If they actually get these proportions, then we say that each process has gotten its “fair share” of the CPU. A stricter interpretation of fair share is that every process gets the *same* fraction of CPU time; if there are n processes, each should get $1/n$ of the CPU time, again, over some specified period of time. We will use the broader interpretation.

For example, say that process A is to get 50% of the CPU time, process B is to get 10%, and process C is to get 40%. These given amounts may be based on requests by processes, as in when a process begins its execution it may request a certain fraction of CPU time, e.g., 50%. If that request can be satisfied (if 50% of the CPU is available), the process is allowed to run, and the scheduler tries to meet that request over some specified period of time. If that request cannot be satisfied, the process would not be allowed to run, or, it might be given the chance to modify its request.

Another approach might be that the proportions are based on how much a user is willing to pay. To get a higher proportion of the CPU time, one

must pay more. Regardless of how the requests are determined, the question is, can the scheduler properly apportion CPU time to meet those requests? More precisely, which process should get the next quantum so that, over some specified period of time, the proportion of CPU time that each process actually gets is what was requested?

What do we mean by “over a specified period of time.” In general, we can only satisfy meeting the requested proportions if a long enough period of time has occurred. To take an extreme example where this would not be possible, say that the period of time were exactly one quantum. Whichever process were assigned that quantum, it will have gotten 100% of the CPU time, and all other processes would have gotten 0%. This is the only apportionment that is possible over a period of one quantum. If the period is, say, 1000 quantums, then we might specify that the fraction of quantums that a process gets over this time must match what they requested, perhaps with some given acceptable error, such as 1%.

Figure 4.11 shows an example of FS. To implement this policy, at the end of each quantum, we determine what fraction of time the process actually received, and compare this to how much it requested. Let’s assume that all three processes arrive at time 0, so they all begin at the same time, and as indicated above, A has requested 50%, B has requested 10%, and C has requested 40%. Since none have gotten a chance to run, they have all been treated “equally” or “fairly” so far, and so there is no preference as to which process runs first, so let’s choose A.

	1	2	3	4	5	6	7	8	9	10
A	100%	50%	33%	50%	40%	50%	43%	50%	44%	50%
B	0%	50%	33%	25%	20%	17%	14%	13%	11%	10%
C	0%	0%	33%	25%	40%	33%	43%	38%	44%	40%

Figure 4.11: Schedule using Fair Share (FS).

At the end of the first quantum, since process A ran during that quantum, it received 100% of the CPU, while it had only requested 50%. The other two processes, B and C, ran for 0% of the time so far (since they have not run yet), while having requested 10% and 40% respectively. We then compute the ratio of amount received to amount requested. For A, the “received-to-requested” ratio is 2 ($= 100\%/50\%$), and for B and C, the ratio is 0. We then select the process with the smallest ratio to get the CPU next.

In a sense, we are selecting the process that has been treated “least fairly” so far. Since process A has gotten more than it expected, two times more to be precise, we would say it received more than its “fair share,” while the other processes both received less than their fair shares. Both B and C are tied with ratios of 0, so we can select either one to run next. Let’s choose B.

At the end of time 2, we would recalculate all the fractions and ratios. At time 2, since process A ran for one quantum and was waiting for one quantum, the actual fraction of CPU time it received is 50%. Since process B also ran for one quantum and was waiting for one quantum, the actual fraction of CPU time it received is also 50%. Since process C did not run,

the actual fraction of CPU time it received a 0%. We compute the received-to-requested ratios for each one, $50\%/50\% = 1$ for A, $50\%/10\% = 5$ for B, and $0\%/40\% = 0$, for C, and we see that C has the smallest ratio. It was treated “least fairly.” Consequently, it is given the CPU for one quantum.

At the end of time 3, we again recalculate the fractions and ratios. At time 3, since process A ran for one quantum and was waiting for two quantums, the actual fraction of CPU time it received is now 33%. Since process B also ran for one quantum and was waiting for two quantums, the actual fraction of CPU time it received is also 33%. Similarly for C, since it ran for one quantum and was waiting for two quantums, the actual fraction of CPU time it received is also 33%. We compute the received-to-requested ratios for each one, $33\%/50\% = 0.67$ for A, $33\%/10\% = 3.3$ for B, and $33\%/40\% = 0.83$, for C, and we see that A has the smallest ratio. It is now A that was treated least fairly. Consequently, it is given the CPU at time 3, for one quantum.

The fractions shown in Figure 4.11 represent the fraction of time each process has received so far. The smallest received-to-requested ratio determines the decision as to which process should run in the next quantum. After 10 time units, we can see that the actual amounts received correspond precisely to the amounts requested, 50% for A, 10% for B, and 40% for C. And so, over a period of 10 time units, we see that in this example, the scheduler is able to meet the requests precisely as given. At all prior times, the amounts received do not precisely match the amounts requested. Thus, we see that it takes some time for the statistics to begin working out to approach the

target requested fractions.

The amount of computing involved to determine which process should run, as we've described FS so far, is quite a lot. It may not seem that way because we've been discussing only three processes. If we had thousands of processes, and at the end of each quantum, we have to do the calculations as described (which include relatively expensive divisions), that would start amounting to a lot of overhead. Fortunately, there is an algorithm that accomplishes the same goal, with a minimal amount of work, which we discuss next.

4.11 Stride Scheduling

In *Stride Scheduling* (SS), assume there are a set of processes A, B, C, ..., and each has made a request, R_A, R_B, R_C, \dots that correspond to fractions of CPU time. We then calculate *stride values*, which are the reciprocal of the request values, $S_A = 1/R_A, S_B = 1/R_B, S_C = 1/R_C, \dots$. Finally, for each process X, the algorithm maintains a *pass value* P_X , initialized to zero.

At the beginning of each time unit, the scheduler selects the process with the smallest pass value (if there is a tie, then any process that is part of the tie can be selected). That process will then run for one time unit. At the end of the time unit, that process's pass value is incremented by its stride value, $P_X = P_X + S_X$. To decide which process should get to run in the next time unit, the scheduler will repeat what was just described: it will select

the process with the smallest pass value, allow it to run for one time unit, and then increment its pass value by the stride value.

Figure 4.12 shows an example with three processes, A, B, and C, each having requested 50%, 10%, and 40% of CPU time, represented as integers 50, 10, and 40, respectively. Rather than taking their inverses, which will result in floating-point numbers, we can take a very large number L, say 100000, and divide L by the request values to arrive at integer stride values 2000, 10000, and 2500. The pass values are all initialized to zero, and since all the processes are tied, we can randomly choose process A to run in the first time unit.

Process x	Runs Now	A	B	C	To Run Next
Requested Utilization: R_x (in %)		50	10	40	
Stride: $S_x = L/R_x$ ($L = 100000$)		2000	10000	2500	
Pass: $P_x = P_x + S_x$; init = 0		0	0	0	A
quantum 1					
quantum 2					
quantum 3					
quantum 4					
quantum 5					
quantum 6					
quantum 7					
quantum 8					
quantum 9					
quantum 10					

Figure 4.12: Initialization of Stride Scheduling (SS).

Figure 4.13 shows the result after having run for one time unit, reaching time 1. A's pass value is now 2000, while it is 0 for both B and C. Since B

and C are tied, we can select either one, and so we randomly choose B.

Process x	Runs Now	A	B	C	To Run Next
Requested Utilization: R_x (in %)		50	10	40	
Stride: $S_x = L/R_x$ ($L = 100000$)		2000	10000	2500	
Pass: $P_x = P_x + S_x$; init = 0		0	0	0	A
quantum 1	A	2000	0	0	B
quantum 2					
quantum 3					
quantum 4					
quantum 5					
quantum 6					
quantum 7					
quantum 8					
quantum 9					
quantum 10					

Figure 4.13: Stride scheduling after one time unit.

Figure 4.14 shows the result after having run another time unit, reaching time 2. A's pass value is still 2000, B's pass value is now 10000, while C's pass value is still 0. Since C has the smallest pass value, it is chosen to run next.

Figure 4.15 shows the result after having run another time unit, reaching time 3. A's pass value is still 2000, B's pass value is still 10000, and C's pass value is now 2500. Recall that the way the new pass value is determined is by incrementing it by the stride value, and this is only done for the process that just ran. Since A now has the smallest pass value (2000, vs. 10000 for B and 2500 for C), A is chosen to run next.

Figure 4.16 shows the result after having run for ten time units. A green

Process x	Runs Now	A	B	C	To Run Next
Requested Utilization: R_x (in %)		50	10	40	
Stride: $S_x = L/R_x$ ($L = 100000$)		2000	10000	2500	
Pass: $P_x = P_x + S_x$; init = 0		0	0	0	A
quantum 1	A	2000	0	0	B
quantum 2	B	2000	10000	0	C
quantum 3					
quantum 4					
quantum 5					
quantum 6					
quantum 7					
quantum 8					
quantum 9					
quantum 10					

Figure 4.14: Stride scheduling after two steps.

Process x	Runs Now	A	B	C	To Run Next
Requested Utilization: R_x (in %)		50	10	40	
Stride: $S_x = L/R_x$ ($L = 100000$)		2000	10000	2500	
Pass: $P_x = P_x + S_x$; init = 0		0	0	0	A
quantum 1	A	2000	0	0	B
quantum 2	B	2000	10000	0	C
quantum 3	C	2000	10000	2500	A
quantum 4					
quantum 5					
quantum 6					
quantum 7					
quantum 8					
quantum 9					
quantum 10					

Figure 4.15: Stride scheduling after three time units.

box means that the process ran for that time unit, while a yellow box means that it was not using the CPU. We can count the number of green boxes for A, B, and C, and we see that A ran for 5 time units, B ran for 1 time unit, and C ran for 4 time units, all over a 10 time-unit period. Computing the fraction of time that each process ran, we got 50%, 10%, and 40% for A, B, and C, respectively, precisely what each process requested. The Stride Scheduling algorithm is guaranteed to come as close as possible to giving each process its requested fraction of CPU time at each point in time.

Process x	Runs Now	A	B	C	To Run Next
Requested Utilization: R_x (in %)		50	10	40	
Stride: $S_x = L/R_x$ ($L = 100000$)		2000	10000	2500	
Pass: $P_x = P_x + S_x$; init = 0		0	0	0	A
quantum 1	A	2000	0	0	B
quantum 2	B	2000	10000	0	C
quantum 3	C	2000	10000	2500	A
quantum 4	A	4000	10000	2500	C
quantum 5	C	4000	10000	5000	A
quantum 6	A	6000	10000	5000	C
quantum 7	C	6000	10000	7500	A
quantum 8	A	8000	10000	7500	C
quantum 9	C	8000	10000	10000	A
quantum 10	A	10000	10000	10000	... will repeat

Figure 4.16: Stride scheduling after ten time units.

4.12 Real-Time Scheduling

We have seen scheduling policies that focus on minimizing the average turnaround time, we've seen a policy that supports externally derived priorities, and we've seen policies that seek to give processes a fair share of the CPU. We now move to two scheduling policies that seek to take real time into account, specifically to support the idea the processes meet given deadlines to carry out their computations.

In a *real-time system*, the correctness of a computation depends on both its *logical* result, as well as the *timing* of the result, i.e., whether the result was produced within a certain amount of time. There are generally two types of real-time systems, *hard* real-time and *soft* real-time systems. A *hard real-time system* is one where all of the deadlines absolutely must be met, as if any are not, it might result in a critical failure. An example of a hard real-time system is one that controls a nuclear power plant, where processes must carry out their operations within fixed periods of time, otherwise a catastrophic failure may occur.

A *soft real-time system* is one where it is permissible to miss some number of deadlines. For example, a video player must be able to display a new frame of video every 33 milliseconds (for video consisting of 30 frames per second). However, if a frame is missed every so often, it will not lead to a critical failure; the user will tolerate this, as long as it doesn't happen too often.

We make this distinction because hard real-time systems require the pre-

allocation of a large number of resources in order to strictly meet their stringent deadlines. If we implemented a video player as a hard real-time system, then we would have to pre-allocate so many resources, in terms of CPU time and memory space, that there may be little left to be able to run other processes, which would be very undesirable for the user. Consequently, the user would rather have slightly imperfect video while being able to run other applications, rather than perfect video and not be able to use their computer for anything else.

Real-time systems can also be characterized as to whether they are *periodic* or *aperiodic*. In a *periodic real-time system*, the deadlines occur periodically (after a fixed known amount of time, repeatedly), and can be anticipated. In an *aperiodic real-time system*, deadlines can occur at any time.

We will consider two real-time scheduling policies: Earliest Deadline First and Rate Monotonic Scheduling. To set things up, let's consider an example where we have two periodic processes, and for each process we're given two parameters, C and T , where C is the *CPU burst* and T is the *period*. Within every period of T time units, C time units of CPU time must be allocated to the process to meet its deadline requirement (every T time units represents a new deadline). From C and T , we can compute the utilization, $U = C/T$, which tells us the fraction of time CPU that will be used. For example, if a process requires 40 time units (this is C) every 120 time units (this is T), then the utilization is about 33% (this is U).

We begin with the simple question: Given two processes with parameters C and T for each one, can the processes be ordered in a fixed way such that all the deadlines are met? For example, if we have two processes A and B, if we were to order the processes as A first and then B, and repeat, will all the deadlines be met? Or, with the alternative order, where B went first and then A, and repeated, will all the deadlines be met?

Figure 4.17 shows this example with two processes A and B, along with their parameters C and T and the derived utilizations U . Process A must get $C_A = 15$ time units of CPU time every $T_A = 30$ time units, and process B must get $C_B = 10$ time units of CPU time every $T_B = 20$ time units. Is it possible to order the processes to run such that every 30 time units, process A will get 15 time units of the CPU, and independently, every 20 time units, process B will get 10 time units of the CPU? We first note that sum of the utilizations $U_A + U_B$ cannot exceed 100% (as we cannot give away more CPU time than there is). In this example, the sum of utilizations equals 100%, and so we have not exceeded the maximum amount of available CPU time. But we still don't know whether the deadlines can actually be met.

	C	T	U
A	15	30	50%
B	10	20	50%

Figure 4.17: Two periodic processes.

Figure 4.18 shows the schedule where we run A first and then B, and repeat. We see that B will miss all of its deadlines. A will run for 15 time units and then B will run for 10. Notice that B immediately misses its first deadline, which occurs at time 20. B was not able to receive a full 10 time units of CPU time before time 20, and so this schedule fails.

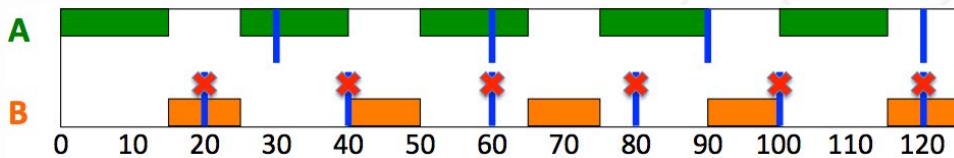


Figure 4.18: Schedule of periodic processes where A runs first, then B.

Figure 4.19 shows the schedule where we run B first and then A, and repeat. In this case, both processes meet their first few deadlines, but process B misses its deadline at time 80. Consequently, we see that neither orderings, ABAB... nor BABA..., work. Is it possible that there is some other schedule that would allow these processes to meet all their deadlines?

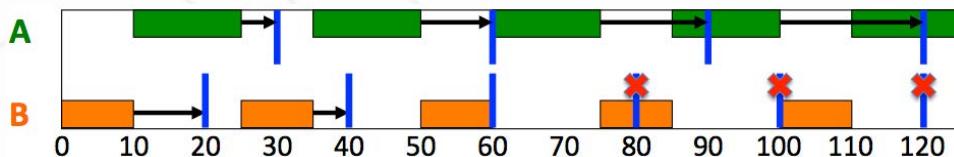


Figure 4.19: Schedule of periodic processes where B runs first, then A.

4.13 Earliest Deadline First

We first consider the *Earliest Deadline First* (EDF) scheduling policy. In EDF, the rule is that we schedule the process with the earliest deadline before any of the others. If a process with an earlier deadline appears, we preempt the currently running process and give the CPU to that process. This algorithm will meet all deadlines and works for both periodic and aperiodic processes. It achieves 100% utilization, *if* we ignore overhead.

EDF is very powerful, but this theoretical result is somewhat weakened in that the overhead is significant. Every time a new deadline appears, we have to compare it to the other deadlines to see where it should fit. Essentially, the algorithm is continually sorting all of the deadlines, so that it always knows which is the earliest deadline.

Let's look at an example of EDF in action. In Figure 4.20, we're given three processes, with a table showing the CPU burst times and periods (our example is for periodic deadlines). When we add up the utilizations, we see that the sum is 100%, and as long as the sum does not exceed 100%, EDF will work. At time 0, the process with the earliest deadline is C, since its deadline is at time 30, and so it gets to run first. When it uses its required 7.5 units of CPU time, we can calculate its next deadline, which will be at 60 (i.e., 30 beyond the first 30).

Consequently, the process with the earliest deadline is now B, where its deadline is at 40. We allow it to run for its required CPU burst of 10 time

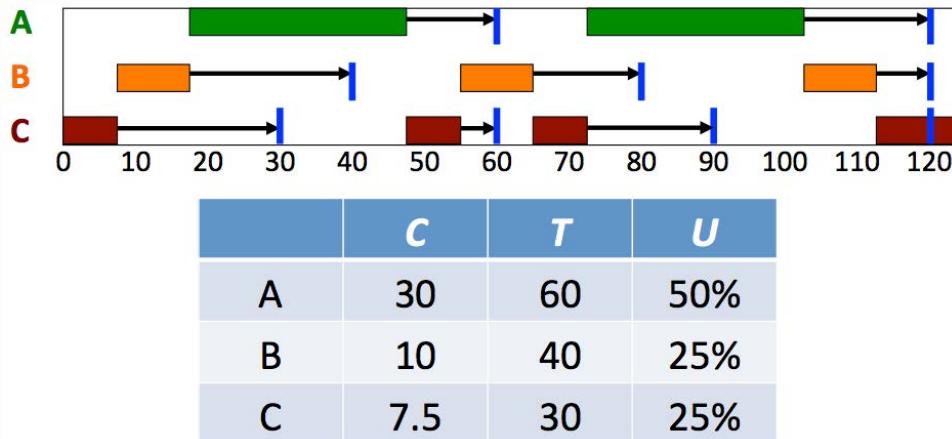


Figure 4.20: Periodic processes scheduled using Earliest Deadline First (EDF).

units, which now satisfies the deadline at 40, and we calculate a new deadline for B, which will now be at 80. The earliest deadline is now at 60; in fact, there are two deadlines at time 60, one for A and one for C. We can select either process to run, and randomly select A which is allowed to run for A's required 30 time units, and then we can calculate its new deadline which will be at 120. This leaves process C, which we let run for 7.5 time units, and it comfortably meets its deadline at 60. C's next deadline will be at 90.

The earliest deadline is now that of process B, which is at 80, and so B is allowed to run for 10 time units. Its new deadline will be at 120. The earliest deadline is now that for process C, which is at 90, and so we allow C to run for 7.5 time units. Its new deadline will be at 120. We now have all three processes that have a deadline at 120, and so we can select any of them to run, just as we did at the start. We first select A, let it run for 30,

then we select B, let it run for 10, and we select C and let it run for 7.5, and we see that all the processes have met all their deadlines. At time 120, the situation is the same as at time 0. All the processes are effectively starting at the same time. And we know that within another 120 time units, all of the deadlines will again be met. And so we can see that EDF meets all deadlines in this example. More generally, EDF will meet all deadlines as long as the sum of the utilizations does not exceed 100%.

Notice how much work had to be done at each step. When a new deadline is calculated, it must be ordered relative to all of the other deadlines so that we can determine the order of the processes and how they should run. And each time a process runs, that order may change due to any deadlines that have been generated. In fact, we can say that every time a new deadline is generated, the amount of work to be done is $O(n)$, where n is the number of processes. It would be desirable if the amount of work to be done is constant, rather than being dependent on the number of processes.

4.14 Rate Monotonic Scheduling

This brings us to the *Rate Monotonic Scheduling* (RMS) policy. RMS only works for periodic processes, where processes will be prioritized based on their rates (the reciprocal of their periods). At the start of a period, the process with the highest priority is selected. When a process runs for its CPU burst, it is put aside until the deadline is reached, after which it is reawakened and

considered for executing during its next period. If that process's priority is greater than the currently running process, the current process is preempted.

RMS provides a guaranty that *all deadlines will be met depending on the following RMS test:*

$$U_1 + U_2 + \dots + U_n = n(2^{1/n} - 1) \quad (4.3)$$

Given n processes having utilizations U_1, U_2, \dots, U_n , if the sum of the utilizations is bounded by the quantity $n(2^{1/n} - 1)$, then the test passes, and RMS guarantees that all the deadlines will be met; if it fails, there is no guarantee.

It is important to be clear on precisely what can and cannot be concluded based on the test. If the test passes, we are able to definitively conclude that all the deadlines will be met, and we can know this *a priori*, prior to the running of the system. It is a remarkable result.

But if the test fails, we cannot conclude anything. It is possible that, as the system runs, a deadline will eventually be missed (and that even others may be missed). But it is also possible that not a single deadline is missed. We are simply not able to say, one way or the other, whether deadlines will be missed or not, and no amount of time running the system will allow us to conclude otherwise (in general).

In Figure 4.21, we're given three processes with their CPU burst times, periods and derived utilizations. From the periods we can compute a rate for

each process, which is simply one divided by the period. We then prioritize the processes according to the rates. In this case, C has highest priority, then B, and then A. If we take the sum of the utilizations, it equals 75%, which passes the RMS test that says the sum must not exceed the bound of $3(2^{1/3}-1) \approx 78\%$. Consequently, we're guaranteed that a schedule where the processes run in the order C, B, A, C, B, A, ... all the deadlines will be met.

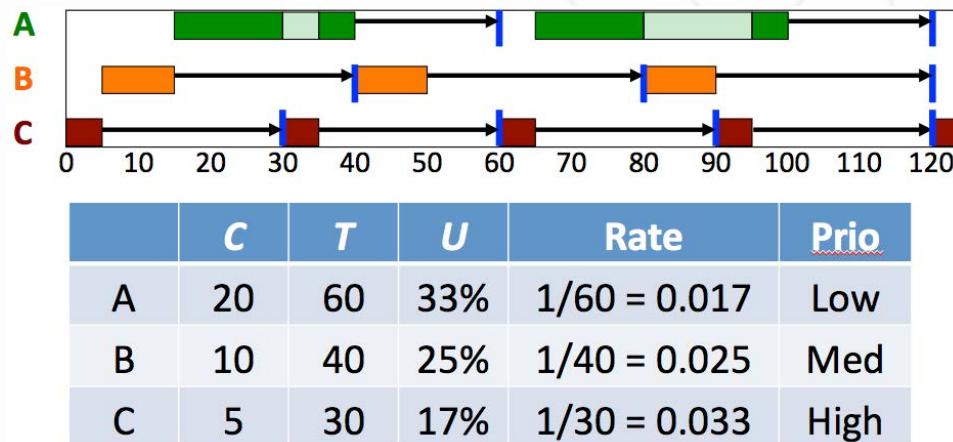


Figure 4.21: Rate Monotonic Scheduling (RMS) with processes' utilizations summing to 75%, which is below the 78% bound, and so all deadlines are guaranteed to be met.

Let's go through the example. At time 0, process C runs for 5 time units, which meets its deadline at time 30. C will not have to run until time 30, and so it is taken out of consideration, and the next higher priority process, which is B, is run for 10 time units, and so it meets its deadline, which is at 40. We can now put process B aside as we will not have to consider it until time 40. The only process that remains is process A, which begins running

for its required 20 time units.

However, at time 30, process C awakens, because its first deadline just passed, and so C is eligible to run to meet its required 5 units of CPU time which must occur before its next deadline at time 60. Since process C has a higher priority than the currently running process, A, A is preempted and C executes for 5 time. When process C completes, process A can resume because B will not reawaken until time 40. From the graph, we can see that all deadlines are met. And, in fact, since the RMS test passes, we are guaranteed that all deadlines will be met.

Now consider the situation in Figure 4.22. With this set of burst times and periods, the sum of the utilizations is 97%, which exceeds the 78% bound, and so there's no guarantee that all the deadlines will be met. And in fact, we see that at time 60, A will miss a deadline.

In Figure 4.23, the sum of the utilizations is 84%, which exceeds the bound of 78%, and so there's no guarantee that the deadlines will be met. However in this example, it turns out that all the deadlines will be met, but there is no way to know that beforehand. All RMS can guarantee is that deadlines will be met if the test passes, but nothing can be said if the test fails.

Finally, in Figure 4.24, we again have a situation where the sum of the utilizations again exceeds the bound, and so the RMS test fails. However, it turns out that all the deadlines will in fact be met, as can be seen in the diagram.

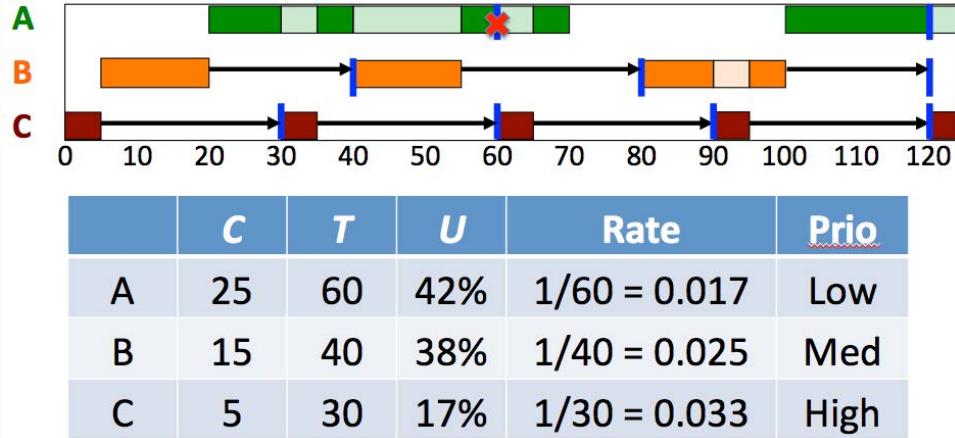


Figure 4.22: RMS with processes' utilizations summing to 97%, which is above the 78% bound, and so there is no guarantee the deadlines will be met, and in fact, a deadline is missed.

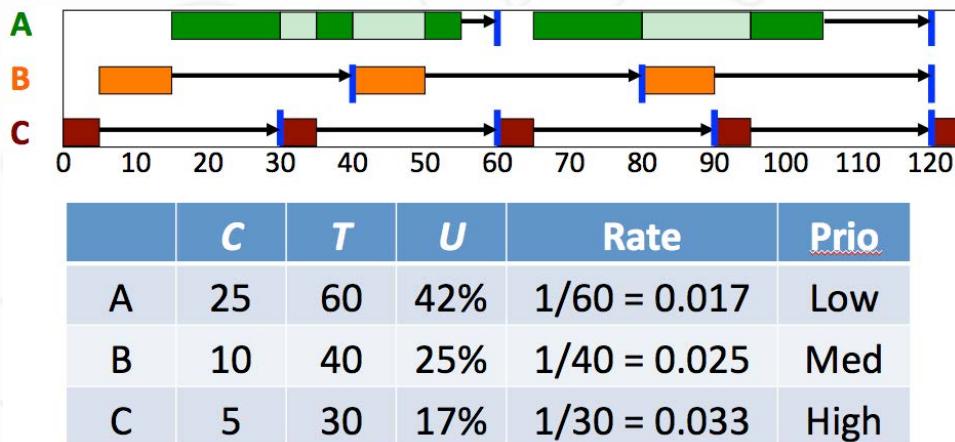


Figure 4.23: RMS with processes' utilizations summing to 84%, which is above the 78% bound, and so there is no guarantee the deadlines will be met; however, they are met.

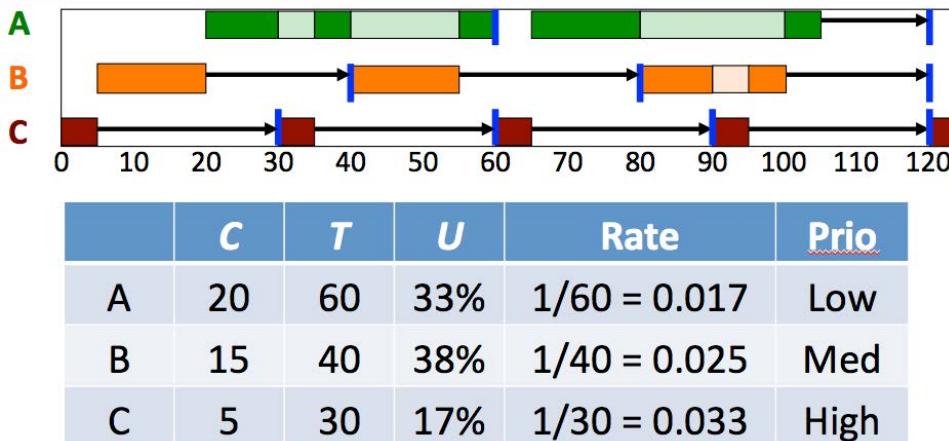


Figure 4.24: RMS with processes' utilizations summing to 88%, which is above the 78% bound, and so there is no guarantee the deadlines will be met; however, they are met.

We can conclude the following about RMS. RMS is optimal, although limited. RMS is simple and efficient, because it is based on *static priority scheduling* and uses predetermined rates. In fact, RMS is *optimal* for static priority algorithms. What this means is that if RMS cannot schedule a set of processes, then *no other static priority algorithm will be able to*.

However, RMS is limited in what guarantees. The overall utilization cannot exceed the $n(2^{1/n}-1)$ bound; if that bound is exceeded, then there is no way of telling whether RMS is able to meet all deadlines. We can determine the minimum value of the bound by letting n go to infinity, resulting in the value $\ln 2$, or about 69%. This tells us that if the sum of the utilizations is below 69%, RMS is guaranteed to meet all the deadlines without even having to calculate the actual bound, which would certainly be greater than

the minimum for any value for n . And finally, we note that RMS is limited to periodic processes, whereas EDF also supports aperiodic processes.

4.15 Summary

We reviewed a number of CPU scheduling policies, and we noted that the best one will depend on the goals of the system. First Come First Served (FCFS) is a very simple non-preemptive algorithm, but has limited performance. Round Robin (RR) is also simple, but it requires preemption, and delivers good performance. We looked at two benchmark algorithms, Shortest Process Next (SPN) and Shortest Remaining Time (SRT), the first being non-preemptive and the second being preemptive, each of which order processes by the shortest service time. We saw that Multi-Level Feedback Queues (MLFQ) is a policy that uses multiple queues to adaptively determine which processes are shorter and which processes are longer, resulting in very good performance that seeks to approximate Shortest Remaining Time. We looked at Priority Scheduling, which allows processes to be scheduled according to external criteria. We looked at the Fair Share (FS), or Proportional Share algorithm, which seeks to allocate the CPU according to predetermined fractions of time. Finally, we looked at two real-time scheduling algorithms, Earliest Deadline First (EDF) and Rate Monotonic Scheduling (RMS), and saw that EDF meets all deadlines but has significant overhead, while RMS meets all deadlines if the sum of utilizations is appropriately bounded, but

unlike EDF, is very efficient.

4.16 Exercises

1. What is the CPU scheduling problem?
2. What are various possibilities for how much CPU a process should get when it runs?
3. Is there a best scheduling policy?
4. What are various goals that might affect the scheduling policy, and for each goal, what would the effect be?*
5. What are two goals that conflict, and in what way would they conflict?*
6. What is meant by “arrival time”?
7. What is meant by “service time”?
8. What is meant by “turnaround time”?
9. What is a CPU “burst”?*
10. Can the service time ever be less than the turnaround time, and if so, why?*
11. Can the turnaround time ever be less than the service time, and if so, why?*

12. How is average turnaround time computed?
13. If all processes arrive at the same time but have different service times, what order minimizes average turnaround time?
14. Is there an order that maximizes average turnaround time, and if so, what is it?*
15. Which is better: minimizing or maximizing the average turnaround time?
16. Given 3 processes A, B, and C, that arrive at the same time, and have service times x , y , and z , respectively, can you provide an intuitive proof as to why they should run in the order A, B, and C, if $x < y < z$ in order to minimize average turnaround time?*
17. If processes arrive at different times, how does this affect which order is best for minimizing turnaround time?**
18. What is meant by a “non-preemptive” scheduling policy?*
19. What is meant by a “preemptive” scheduling policy?*
20. What are the pros and cons of each?**
21. What mechanism does a preemptive scheduler depend on?**
22. What is meant by “starvation”?*
23. What is FCFS?

24. Is it preemptive or non-preemptive and why?*
25. What are the pros and cons of FCFS?
26. Are there certain processes that especially suffer under FCFS, and why?*
27. Are there certain processes that especially benefit under FCFS, and why?**
28. What is RR?
29. Is it preemptive or non-preemptive and why?*
30. What hardware mechanism is needed for RR?
31. What are the pros and cons of RR?
32. Are there certain processes that especially suffer under RR, and why?*
33. Are there certain processes that especially benefit under RR, and why?**
34. What is SPN?
35. Is it preemptive or non-preemptive and why?*
36. What are the pros and cons of SPN?
37. Are there certain processes that especially suffer under SPN, and why?*
38. Are there certain processes that especially benefit under SPN, and why?**

39. How might starvation occur with SPN?**
40. What is SRT?
41. Is it preemptive or non-preemptive and why?*
42. What are the pros and cons of SRT?
43. Are there certain processes that especially suffer under SRT, and why?*
44. Are there certain processes that especially benefit under SRT, and why?**
45. How might starvation occur with SRT?**
46. What is MLFQ?
47. Is it preemptive or non-preemptive and why?*
48. What are the pros and cons of MLFQ?
49. Are there certain processes that especially suffer under MLFQ, and why?*
50. Are there certain processes that especially benefit under MLFQ, and why?**
51. How might starvation occur with MLFQ?**
52. Under MLFQ, say that there are 3 queues, and each is occupied by a single process: what would be their order of execution?**

53. Under the same scenario, say a new process arrives at time 2- (i.e., a moment before 2); how does this affect the schedule, and what would be the average turnaround time?**
54. Under the same scenario, say a new process arrives at time 5-; how does this affect the schedule, and what would be the average turnaround time?**
55. Under the same scenario, say a new process arrives at time 2- and another arrives at 5-; how does this affect the schedule, and what would be the average turnaround time?**
56. Under MLFQ, what is the rationale for periodically boosting all process priorities?**
57. Under MLFQ, what is the rationale for increasing the number of queues?***
58. Is having more queues always better, why or why not?***
59. What is Priority Scheduling (PS)?
60. Is it preemptive or non-preemptive and why?*
61. What are the pros and cons of PS?
62. Are there certain processes that especially suffer under PS, and why?*
63. Are there certain processes that especially benefit under PS, and why?**
64. How might starvation occur with PS?**

65. Say that PS is implemented such that priority is set to $1/CPU_time_used$: what goal does this scheduling policy try to achieve?***
66. Say that PS is implemented such that priority is set to $time_waiting_for_CPU$: what goal does this scheduling policy try to achieve?***
67. What is Fair Share scheduling (FS)?
68. Is it preemptive or non-preemptive and why?*
69. What are the pros and cons of FS?
70. Are there certain processes that especially suffer under FS, and why?*
71. Are there certain processes that especially benefit under FS, and why?**
72. How might starvation occur with FS?**
73. What is meant by “utilization”?
74. What is the goal of FS in terms of utilization?
75. When are scheduling decisions made in FS?*
76. How is a process selected for FS?**
77. In Figure 4.11, can you explain how each and every percentage is determined in the chart?**
78. How do we know that FS achieved its goal?**

79. What is the Stride Scheduling (SS) algorithm used for?**
80. What are the inputs to SS?
81. What is meant by a “stride” value?*
82. What is meant by a “pass” value?*
83. How often is a scheduling decision made in SS?**
84. What calculation is the decision based on?*
85. What is the purpose of using a large value like 100000 as the numerator in determining stride values?**
86. Why is it important to make the numerator large?***
87. Conversely, what might happen if the numerator were too small?***
88. In Figure 4.16, can you explain each and every step, and how the numbers are calculated?**
89. In the example in Figure 4.16, did SS achieve its goal, and why?**
90. What is real-time scheduling?**
91. How does real-time scheduling differ from the previous scheduling algorithms?**
92. What are the various types of real-time scheduling algorithms, and can you give an example for each one?**

93. What are the pros and cons of each type of real-time scheduling algorithm?***
94. What is a “periodic process”?*
95. What is meant by a “CPU burst”?
96. What is meant by a “period”?
97. How is the utilization calculated, and why is it calculated this way (why use those parameters, and why in a ratio)?**
98. In Figure 4.18, are all the deadlines met, and if not, why?**
99. In Figure 4.19, are all the deadlines met, and if not, why?**
100. What is the point of having shown the schedules in Figure 4.18 and Figure 4.19?***
101. What is EDF?
102. How does EDF work?**
103. What are the pros and cons of EDF?*
104. In Figure 4.20, can you explain the schedule shown, and for each scheduling decision, why the particular choice of process was made?**
105. How does RMS work?**
106. What are the pros and cons of RMS?*

107. In Figure 4.21, can you explain the schedule shown, and for each scheduling decision, why the particular choice of process was made?**
108. How does RMS compare to EDF; in what ways is it better, and in what ways is it worse?**
109. How is the RMS test used?**
110. In Figure 4.22, can you explain the schedule shown, and for each scheduling decision, why the particular choice of process was made?**
111. What is the point of Figure 4.22?**
112. In Figure 4.23, can you explain the schedule shown, and for each scheduling decision, why the particular choice of process was made?**
113. What is the point of Figure 4.23?**
114. In Figure 4.24, can you explain the schedule shown, and for each scheduling decision, why the particular choice of process was made?**
115. What is the point of Figure 4.24?**
116. What is meant by “static priority scheduling”?***
117. Is RMS a static priority scheduling algorithm, and if so why, or why not?***
118. Is EDF a static priority scheduling algorithm, and if so why, or why not?***

119. Is RMS an optimal static priority scheduling algorithm, and if so why, or why not?***
120. What does it mean that in RMS, the utilization test is bounded by
 $\ln 2 \approx 69\%**$
121. Under RMS, if the RMS test passes (or RMS constraint is met), does this guarantee all deadlines will be met?**
122. If the RMS test fails (or RMS constraint is not met), does this guarantee all deadlines will not be met?**

Chapter 5

Synchronization

In the previous chapters, we discussed the process abstraction, and the mechanisms and policies to realize this abstraction. We now consider how processes might interact, and what type of support is needed to allow for such interaction.

We use the term *synchronize* to indicate when events happen *at the same time*. We can apply the concept of synchronization to processes: events in processes that occur “at the same time” are considered synchronized. However, when we consider that a system may have a single CPU, and so only one process can really be making actual progress at any point in time, we use the term synchronization to indicate when one process *waits* for another to reach a certain point. Such a point may be the occurrence of an *event* during the process’s execution, such as the execution of a particular instruction. Thus, if two processes have events that are synchronized, i.e., that are to

occur at the same time, when one process reaches that event, it is then made to wait until the other process reaches the corresponding event, so that *effectively*, they occur “at the same time.”

There are various uses of synchronization. One is to prevent *race conditions*. Another is to allow a process to wait for resources to become available. Synchronization is a key requirement, and it is up to the operating system to provide support for processes so that they may synchronize when necessary.

5.1 The Credit/Debit Problem

We motivate the need for synchronization by considering the following well-known problem, called the *Credit/Debit Problem*. Say that you have \$1000 dollars in your bank account. You now deposit \$100 dollars, and then you withdraw \$100. How much should you have your account? Of course, the answer is that you expect to still have your original \$1000.

But what if the deposit and withdrawal were somehow done *simultaneously*? Imagine that you and a friend arranged to go to two different ATM machines, and one of you made a deposit while your friend made a withdrawal. Furthermore, both of you were able to somehow time this perfectly so that the deposit and withdrawal occurred at the same time. What might we expect would happen?

In Figure 5.1, we show two processes, P_0 and P_1 , where P_0 is running the credit program and P_1 is running the debit program. They are each

passed the value \$100. Let's assume that both processes are running on the same computer, and that there is a single CPU. Consequently, only one of the processes can actually be executing on the CPU at any point in time, and that there is context switching between the processes. Assume that, as discussed earlier, a deposit and withdrawal, both for \$100, have been submitted, and that the two processes are trying to process these requests.

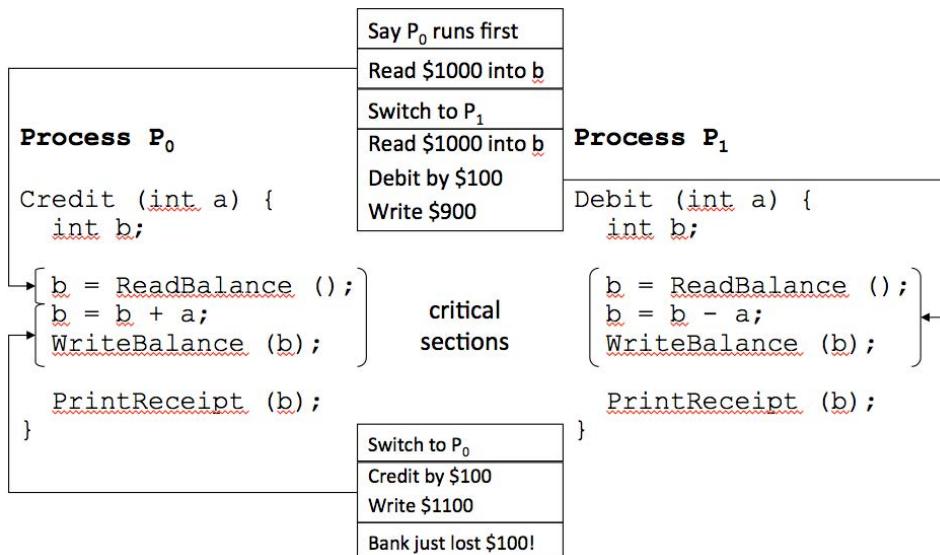


Figure 5.1: The Credit/Debit problem.

Say that P_0 happens to run first, so it is running on the CPU, and the first thing it does is call `ReadBalance`, which is a database method that reads the current balance of \$1000 from the database and places it into the local variable b (which is in the memory of P_0). At this point, assume there is a context switch to P_1 . The debit process begins executing, calls `ReadBalance`,

which reads \$1000 (the database has not been written to, so the balance is still recorded as \$1000), and the result is stored in P_1 's local variable b . It then subtracts a from b , where $a = \$100$, which is the amount that was passed to P_1 when it began executing. The variable b , as recorded in P_1 's memory, is now equal to \$900, and that is the value that is written out to the database to record that the balance is currently \$900.

Now, assume there is a context switch to P_0 . P_0 had just read the balance of \$1000 into its local variable b , which has not changed, and it now increments b by the amount a , which is equal to \$100, as this corresponds to the amount that was initially passed to P_0 , i.e., the amount by which the account should be credited. Consequently, b is now equal to \$1100, and that balance is written out to the database. This overwrites the old value of \$900. So, you just made \$100 and the bank lost \$100!

Clearly this is not a good situation (for the bank, at least!). There should be a way to coordinate the processes so that this kind of problem cannot occur. We call this problem a *race condition*, as the final result depends on a race as to which process is the first (or the last) to execute various statements. It could have just as well occurred that the ending balance was \$900 or \$1000. We would like the processes coordinated so that the result is always what is to be expected, which should be \$1000. How can this problem be resolved?

5.2 Critical Sections

If we look at the code for `credit` and `debit`, we see that there are a number of statements that, when they are interleaved in a certain way in their execution, they could lead to these ambiguous results. If we, as the programmer, were able to constrain the order in which those statements can be executed, we could prevent this ambiguity. These sections of code that could lead to race conditions are called *critical sections*. If we could constrain critical sections to execute such that they cannot be interleaved with respect to each other, we could avoid the problem.

To avoid race conditions, it is the programmer that must identify the critical sections. As shown in Figure 5.2, these sections of code must then run *atomically*, i.e., indivisibly, with respect to each other. Imagine that we did not permit a context switch to occur while a process was in its critical section. In that case, it would be impossible for the statements of two critical sections to become interleaved.

Let's look more closely at the term, "atomic." Atomic means indivisible, and in this context, means that the execution of the critical section will not be divided or broken up in parts such that code in another process's critical section will run. One way to achieve this is to prevent context switches (which we can easily do by turning off the clock interrupt), but this solution is not generally a good one, as it is too drastic.

Consider a process that is running in its critical section, and that there

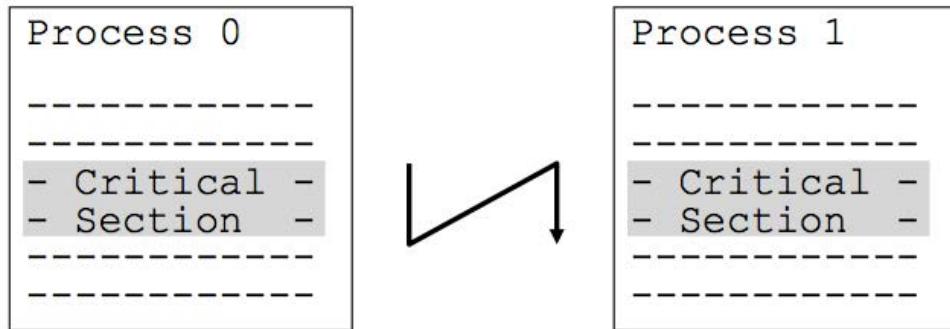


Figure 5.2: Two processes with related critical sections, each of which must run atomically relative to the other.

is a context switch to another process which is *not* running in its critical section. Since damage only occurs when the statements of critical sections are interleaved, this situation would not cause a problem, and so we should allow this situation.

Consequently, what we want is what we might call *effective atomicity*, by which we mean having the same effect as atomicity, regardless as to how it is achieved. We can allow context switches, as long as they don't cause problems. If the context switch is to another process that is not executing critical section code, this is OK. In fact, we certainly would want to allow context switches to processes that are unrelated, and thus will have no effect on the current process (even if it has its own critical section of code, but which is unrelated to that of the current process).

To achieve this, we need some help from the programmer; the operating system cannot figure out on its own whether a context switch will cause a problem or not, and so the idea is to allow context switches, but have some

other way of enforcing atomicity between related critical sections.

5.3 Mutual Exclusion

To avoid race conditions, we need to enforce *mutual exclusion*. By this, we mean that only one process can be active in a critical section at a time. It is okay for another process to execute, as long as that process does not go in its critical section. (In this discussion, we will be assuming that the critical sections of various processes that we discussed are related to each other. It is certainly possible, and indeed very likely, that there will be processes that have critical sections that are unrelated, i.e., whose statements do not effect each other and so how they are interleaved does not matter, and in that case it is not necessary to enforce mutual exclusion between processes that have unrelated critical sections. To simplify our discussion, we will assume that the only processes in existence are ones whose critical sections are related, and so we must enforce mutual exclusion between them.)

How is it possible to achieve mutual exclusion? The goal here is for the operating system designer to provide the programmer with primitives to surround critical sections of code with additional *entry code* and *exit code*, as shown in Figure 5.3. The entry code should act as a barrier. If another process is in the critical section, then the current process should be blocked by the entry code to prevent it from entering the critical section; otherwise, we would not have mutual exclusion. If there is no other process in their

critical section, then the current process should be allowed to proceed by its entry code.

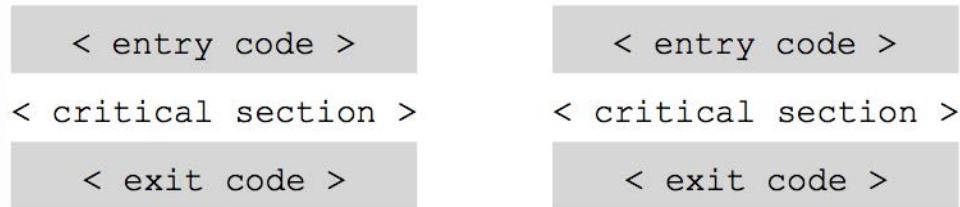


Figure 5.3: Critical sections are protected by entry and exit code to enforce mutual exclusion. How should the entry and exit code be designed?

The exit code essentially acts as a signal to indicate that the current process is done with its critical section. Consequently, if any other process was blocked because it was attempting to enter its critical section and became blocked, it should now be unblocked.

Notice that the programmer and operating system designer have two distinct roles. It is the role of the operating system designer to provide the programmer with the primitives that support mutual exclusion. It is the role of the programmer to identify the critical sections, and make use of the primitives so that mutual exclusion is enforced.

5.4 Requirements for Mutual Exclusion

There is a set of requirements that determine whether a solution is a good one for mutual exclusion. Given a set of multiple cooperating processes, each with related critical sections, the following requirements must be met:

1. There can be at most one process in a critical section at any point in time
2. A process cannot be prevented entry into their critical section if no other processes are in their critical sections
3. A process must eventually be allowed to enter its critical section (i.e., it cannot be made to wait forever)
4. No assumptions can be made regarding the speed of the CPU or the number of CPUs

To elaborate on these requirements, let's look at a series of examples and see which one of the requirements are met and not met. In each example, there are two processes, P_0 and P_1 , which are to run in parallel. We cannot make any assumptions about when context switches can occur; they can occur any time, and so the processes' respective statements can be interleaved in many ways. We can presume a preemptive scheduler where, after a certain amount of time (a quantum), the current process is preempted and the other is allowed to run. But the code cannot be written to take advantage of this knowledge as to how long a quantum is (it would be very difficult to do so anyway); we must assume a context switch can happen at any time. We must find the ways that might cause problems, and see how to avoid them. This is generally a very difficult exercise.

In Figure 5.4, we have two processes, P_0 and P_1 . They both have critical sections (for which we want that they run atomically relative to each other),

and the critical sections are surrounded by entry and exit code, following the pattern shown in Figure 5.3. The question is whether the entry and exit code follows the requirements given above.

```
shared int lock = OPEN;

P0
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;

P1
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

Figure 5.4: Using a software lock.

The processes share a variable called “lock.” By “share a variable,” we mean that, if one process updates the value of the variable, the other process will see the update. Consequently, this is not a variable such as a global or local variable that exist in data or stack areas of a particular process, but rather the variable is in an area of memory that both processes have access to, or share. (We have not yet shown a process memory structure that supports this model; this will have to wait until an upcoming chapter when we focus on memory. For now, simply assume that processes are able to share variables, somehow.)

Assume that P_0 has the CPU. The first thing P_0 does is execute the entry code that is before the critical section, for which the first statement is the `while` statement. The `while` condition is tested, checking whether the value of `lock` equals `CLOSED`. Given that `lock` was initialized to `OPEN`, the condition

is false, and so P_0 goes ahead and executes the next statement, which is to set the value of `lock` to `CLOSED`. It then enters its critical section. While executing in its critical section, let's assume that a context switch occurs and that P_1 begins running. (Note that we, as designers of the entry and exit code, do not know what code is inside the critical section, or how long it might be executing there. It might be momentary, or it might be a very long time, so the possibility of a context switch is one we must consider it, as this is important when trying to determine if mutual exclusion is enforced or not.)

P_1 begins by executing the entry code prior to its critical section, and checks whether `lock` is equal to `CLOSED` in the `while` statement. At this point, `lock` is indeed equal to `CLOSED` (as set by P_0), and so P_1 will simply keep looping, repeatedly checking the `while` condition. Since there is no reason for the value of `lock` to change, it continues doing so until a context switch to P_0 occurs. This repeated looping is effectively causing P_1 to wait, preventing it from entering its critical section; this is called *busy waiting*. It is wasteful, but it achieves the goal of causing P_1 to not enter the critical section.

P_0 runs and eventually exits its critical section, executing the exit code, which sets `lock` equal to `OPEN`. Eventually, there is a context switch back to P_1 , which was stuck in the `while` loop. Since `lock` is now set to `OPEN`, the `while` condition is false, and P_1 can proceed. It first sets the value of `lock` to `CLOSED`, and enters its critical section. Eventually, it will be done with its

critical section, and upon exiting, will set `lock` equal to `OPEN`.

In our sample run, we see that mutual exclusion was enforced. However, is it possible that some other code sequence, due to context switches occurring at other times, might cause a problem? Consider the following scenario. Say that P_0 runs and checks the value of `lock`, and since `lock` is `OPEN`, the `while` condition is false, and so P_0 proceeds to the next statement, just as above. However just before setting `lock` equal to `CLOSED`, assume that there's a context switch to P_1 . P_1 now runs and checks the value of `lock`, which is still equal to `OPEN`, and so the `while` condition is false, and P_1 is able to proceed. It sets `lock` equal to `CLOSED`, and then enters its critical section. So far, everything is OK (as there is only one process in its critical section, namely P_1).

While P_1 is inside its critical section, say a context switch occurs to P_0 . P_0 resumes, and recall that it had already checked whether `lock` was `CLOSED`, and since it wasn't, it was able to proceed. So, it now sets `lock` equal to `CLOSED` (it already is, but P_0 doesn't know that), and enters its critical section. We now have two processes in their critical section; mutual exclusion failed! Consequently, this form of entry and exit code, which relies on a software lock, is not a good solution, as requirement #1 is not met. We need to find a better solution.

In Figure 5.5, we've modified the entry and exit codes to make use of a shared variable called “`turn`.” The idea here is that processes will take turns entering their critical section, and the `turn` variable will indicate which

process is allowed to enter its critical section next. Initializing `turn` to 0 means that it is initially P_0 's turn.

```
shared int turn = 0;           // arbitrary set to P0
P0
while (turn != 0);
< critical section >
turn = 1;
P1
while (turn != 1);
< critical section >
turn = 0;
```

Figure 5.5: Taking turns.

Let's assume that P_0 runs first. The condition in the `while` loop checks whether `turn` is not equal to 0; in other words, if it is not P_0 's `turn`, the condition will be true, and P_0 will busy wait. Given the `turn` is equal to 0, P_0 is allowed to enter its critical section. At this point, let's assume a context switch to P_1 occurs.

P_1 will check whether it is not its `turn`, and indeed, since `turn` is equal to 0, the condition as to whether `turn` is not equal to 0 is true, and so P_1 busy waits. This is precisely the behavior that we want. Eventually, P_1 will use its quantum, and there will be a context switch to P_0 . P_0 will resume executing its critical section, eventually it will exit, setting `turn` equal to 1, and then exit.

P_1 will be resumed, and this time, upon checking whether `turn` is not equal to one, the condition is false, and so P_1 will exit the `while` loop, and enter its critical section. When it completes, it sets `turn` to 0, and exits.

It seems that mutual exclusion is achieved. But, is there problem? In fact, there is. Let's assume that P_1 were to run first. The variable `turn` is initialized to 0, and so P_1 will busy wait because the condition as to whether `turn` is not equal to 1 is true. P_1 is being prevented from entering its critical section. And yet, P_0 is not in its critical section, and so there is no reason for P_1 to be prevented from entering its critical section. This goes directly against requirement #2, which says that a process cannot be prevented entry into their critical section if no other processes are in their critical sections.

So, we must discard this solution.

In Figure 5.6, we devise a more elaborate solution that involves processes “stating their intentions.” Prior to entering a critical section, a process will indicate that it intends to enter its critical section. But before doing so, it checks whether the other process had already indicated its intention to enter. If the other process did, then the current process will busy wait. Otherwise, it will proceed into critical section. Let's see if this works.

```
shared boolean intent[2] = {FALSE, FALSE};

P0
intent[0] = TRUE;
while (intent[1]);
< critical section >
intent[0] = FALSE;

P1
intent[1] = TRUE;
while (intent[0]);
< critical section >
intent[1] = FALSE;
```

Figure 5.6: Stating intentions.

Let's assume that P_0 begins running, and so it sets `intent[0]` to TRUE, which indicates that it intends to enter its critical section. In the `while` loop, it then checks whether `intent[1]` is TRUE; in other words, had P_1 already indicated that it wants to enter its critical section? Since `intent[1]` is FALSE (because this is how it is initialized), P_0 exits the `while` loop, and can proceed into its critical section. Let's now assume that a context switch occurs to P_1 .

P_1 will begin executing and will set `intent[1]` to TRUE, indicating its intention to enter its critical section. It then checks whether P_0 had already indicated an intention to enter its critical section. Since `intent[0]` is TRUE, P_1 will busy wait and thus be prevented from entering its critical section, precisely the behavior we desire. Eventually, P_1 will use its quantum, and there will be a context switch to P_0 . P_0 will resume executing its critical section, eventually it will be done, setting `intent[0]` to FALSE, and then exits.

P_1 will be resumed, but this time, upon checking whether `intent[0]` is TRUE, the condition is false, and so P_1 will exit the `while` loop and enter its critical section. When it completes, it sets `intent[1]` to FALSE, and exits.

As in the earlier examples, mutual exclusion seems to be achieved. But if we've learned anything, it is that validating mutual exclusion can be tricky, and that we need to be careful about reaching any conclusion. And in fact, there is a problem.

Again, we must consider that a context switch can occur at any time. As there are many possibilities as to where a context switch can occur, the

analysis is not easy, and may be cumbersome. To be absolutely sure, we would have to consider every possible interleaving of instructions between P_0 and P_1 , which would be very time-consuming and error prone.

Let's again assume that P_0 begins running, and so it first sets `intent[0]` to TRUE. At this point, assume a context switch to P_1 occurs. P_1 will begin executing and will set `intent[1]` to TRUE. Now, both have indicated as TRUE their intent to enter their respective critical sections. If P_1 continues running, it will busy wait in its `while` loop because P_0 had set `intent[0]` to TRUE. Eventually, there will be a context switch to P_0 , and P_0 will busy wait in its `while` loop because P_1 had set `intent[1]` to TRUE.

We see that *neither* process will be able to enter its critical section, which is not the behavior we want. In fact, this behavior breaks requirement #3, which says that a process must eventually be allowed to enter its critical section; it cannot be made to wait forever. And so we will have to discard this solution.

5.5 Peterson's Solution

It seems that every attempt we've made to improve our solution has been met with yet another problem. Is it possible that maybe there is no solution to the mutual exclusion problem? In fact, for a time, computer scientists did not know whether a solution was possible. However, we do have solutions, one of the more popular being *Peterson's solution* (due to Gary L. Peterson,

a professor at the University of Rochester), which we now describe. Its construction includes elements from the previous solutions we've attempted, and puts them together into a working solution.

Figure 5.7 shows Peterson's solution to the mutual exclusion problem. It makes use of a `turn` variable, as well as an `intent` array of booleans where processes indicate their intent to enter their critical sections. Let's see how it works.

```
shared int turn;
shared boolean intent[2] = {FALSE, FALSE};

P0
intent[0] = TRUE;
turn = 1;
while (intent[1] && turn==1);
< critical section >
intent[0] = FALSE;

P1
intent[1] = TRUE;
turn = 0;
while (intent[0] && turn==0);
< critical section >
intent[1] = FALSE;
```

Figure 5.7: Peterson's Solution to the mutual exclusion problem, for two processes.

Let's assume that P_0 runs first. It will first set `intent[0]` to TRUE, and then set `turn` equal to 1, thus giving P_1 priority to enter (as if it were saying, "I would like to enter, but only after you"). P_0 then checks whether P_1 expressed an intent to enter and whether it is P_1 's turn. If both of these conditions are true, then P_0 will busy wait. Otherwise, it will proceed into its critical section.

Assume that a context switch to P_1 now occurs. P_1 will set its intent to

TRUE, and then set turn equal to 0, giving P_0 priority to enter. P_1 then checks whether P_0 expressed its intent to enter and whether it is P_0 's turn. In this case, P_0 did set its intent, and it is P_0 's turn, as set by P_1 . Consequently, P_1 will busy wait, which is correct behavior, so far.

Eventually there will be a context switch to P_0 , and P_0 will resume inside its critical section. When it is done, it sets its `intent[0]` to FALSE, and exits.

P_1 will then run, and since `intent[0] = FALSE`, it breaks out of the while loop, and enters its critical section. When it is done, it sets its `intent[1]` to FALSE, and exits.

For this particular interleaving of code, mutual exclusion is achieved. But we have to at least consider the other interleavings that caused problems to have better confidence that it does work.

Say that P_0 runs first, it sets `intent[0]` to TRUE, sets turn to 1, and there is a context switch to P_1 . P_1 runs, sets `intent[1]` to TRUE, sets turn to 0. At this point, whichever process gets the CPU, one of them, and only one of them, will surely be able to proceed. This addresses requirements #1 and #3. If only one of them were in the system, say P_1 , it would set `intent[1]` to TRUE, set turn to 0, and since `intent[0]` is FALSE, it would be able to enter its critical section, rather than busy wait, despite that P_0 does not exist. This addresses requirement #2.

Our reasoning so far is not a proof, but it does show that the situations that caused problems earlier will no longer be problems. A more formal

proof would consider that when the processes reach their respective `while` statements, regardless of the interleaving of the earlier statements, one and only one process will be able to proceed, and that it will definitely proceed. This satisfies the first three requirements. Furthermore, our reasoning does not make any assumptions about the CPU speed (which means that we are allowing a context to occur anywhere, and so all interleaved orderings of code are possible), nor does it make any assumption about the number of CPUs (which means that there can be true parallelism if there are two CPUs, but this does not affect the reasoning about the `while` loops only allowing one, at least and at most, process to proceed), we can see that requirement #4 is also satisfied.

Actually, for this last requirement, we need the assumption that our physical memory operates atomically, i.e., if two or more memory requests are made “at the same time,” the memory hardware will arbitrate such that they are processed one at a time (this does not say anything about the order in which they are processed; they can be done in any order). Most physical memories work this way, so the requirement about not making any assumptions about the number of CPUs is not affected.

Essentially, Peterson's solution is based on the idea that, if there is competition to enter critical sections, the processes will take turns. Otherwise, one will be able to enter. The solution in Figure 5.7 only works for two processes, but there is a more general version of Peterson's solution that works for n processes, for any (integer) value of $n \geq 2$. We note that the general

version is significantly more complex than the one for two processes. But, what is important is that there is a solution to the mutual exclusion problem, which works and meets the requirements, and it works in the general case.

5.6 Disabling Interrupts

The root of the problem in all the scenarios we have looked at is that context switches can occur unexpectedly. What if we got rid of unexpected context switches, only allowing them to occur at carefully determined points in time? We could achieve this by disabling the clock interrupt whenever we wanted to prevent context switches, and enable it when they are deemed OK to occur. After all, if the clock cannot generate an interrupt at times we cannot control, there can be no uncontrolled context switches. If there are no uncontrolled context switches, there can be no races between processes. And if there are no races, we can achieve mutual exclusion. What's the problem with this reasoning?

The problem is that it breaks requirement #4, which says, in part, that we can make no assumptions regarding the number of CPUs. Disabling the clock interrupt to achieve mutual exclusion will only work on a uniprocessor (a single CPU). If there are multiple CPUs, then interrupts would have to be turned off for each one, which can typically only be done sequentially, and while they are being turned off, some context switching might still occur. So, we would need an atomic version of turning off the clock interrupt for all

CPUs, which is not provided by most machines (as a basic capability, though it might be created using other mechanisms).

But even if it could be done, this solution is too strong and thus too restrictive. By turning off the clock interrupt, no context switching is possible, even amongst processes that do not require mutual exclusion; we should certainly allow context switching between those processes. For the processes that do require mutual exclusion, they only require it for their critical sections (as well as for their entry and exit codes that protect critical sections). Outside of critical sections (and the entry and exit codes), even these processes should be allowed to freely context switch between each other. Consequently, we don't want to prevent context switches when they can cause no harm, and so disabling interrupts is not a desirable solution.

As an aside, if we could safely make the assumption that our machine is a uniprocessor, we might adopt the solution of disabling interrupts by doing so at carefully chosen points in time. These points would be just before and just after executing the entry code, as well as just before and just after executing the exit code. The clock interrupt would be disabled just prior to executing the entry code, and enabled just after executing it. We would do the same for the exit code. In this case, the entry and exit codes would execute atomically, while context switching would be allowed outside those very limited situations, even within critical sections.

For example, if the process is inside its critical section, it would be okay for a context switch to occur to another process that is not executing in its

critical section. We know that the other process would not be executing in its critical section because the entry code prevented this, and we know that mutual exclusion is guaranteed for the entry code (and the exit code), because context switches cannot happen during the entry code's (and the exit code's) execution.

In a sense, we have a *lower level* of mutual exclusion that is maintained for the entry and exit code, and because of this, we derive mutual exclusion for critical sections. In fact, before multiprocessors became prevalent, many operating systems would rely on this “trick,” based on the reasonable assumption at the time that the underlying machine had a single CPU. When multiprocessors became prevalent, these operating systems would have to be rewritten to undo the reliance on the uniprocessor assumption (which was not easy, as it was hard to figure out which parts of code were relying on that assumption!).

5.7 Test-and-Set-Lock

The entry code in Peterson's solution is a bit complicated, and not easy to analyze. And in fact, if we were to look at the more general n -process version ($n \geq 2$) of Peterson's solution, we would see that it is very complicated. Peterson's solution is complicated because it is trying to make do with standard software instructions and take advantage of memory arbitration, and without any extra help, it is about the best we can do. But without any

extra hardware support, it remains a perfectly valid software-only solution to solve mutual exclusion, and we can use it if we must limit ourselves to an all software solution.

However, modern architectures typically offer a single instruction that greatly simplifies the implementation for the entry code for mutual exclusion. It is the **TSL**, or *test-and-set-lock*, instruction (or other instructions that can be used to the same effect; we will limit our discussion to TSL). TSL takes a memory address as an operand, **TSL x**, and does the following: It tests whether the value contained at the memory address **x** is 0, and it also stores the value 1 at that memory address **x**.

Consequently, it actually does two operations, but in a special way: they are carried out so that the two are done atomically. Consequently, there is no possibility of an interruption (that might lead to a context switch) in between. Furthermore, since the memory bus is locked during the two operations, we get atomicity even on a multiprocessor. Even if a hardware interrupt occurs during the instruction, the two operations of the TSL instruction remain indivisible.

Figure 5.8 shows a C procedure implementation of the TSL instruction, under the assumption that the procedure executes atomically. C procedures do not run atomically (at least not without extra help), and so Figure 5.8 is shown simply for clarification, to illustrate the functionality of TSL.

We see that TSL takes a parameter “**addr**,” which is a memory address that will store an integer, and it has a local variable called “**val**.” It first

```

TSL(int *addr)
{
    int val;

    val = *addr;
    *addr = 1;
    return ((val == 0) ? 1 : 0);
}

```

Figure 5.8: The TSL instruction, expressed as a procedure in C, where it is assumed that the procedure (somehow) executes atomically.

records the value at memory address `addr` in `val`, then it stores a 1 in memory address `addr`, and then returns 1 (to signify, TRUE) if `addr` equals 0, otherwise 0 (to signify FALSE).

Figure 5.9 shows a solution to the mutual exclusion problem using TSL. It uses a shared variable called “`lock`” that is initialized to 0. Say that P_0 is running, and executes the `while` statement. The condition of the `while` loop is evaluated as follows. If `lock` is equal to 0, TSL returns TRUE, and the condition, which takes the negation, is FALSE. Consequently, P_0 proceeds into the critical section. Now, say that there is a context switch to P_1 . P_1 will execute the `while` statement, and the condition will evaluate to TRUE! The reason is that the value of `lock` will be equal to 1. This is the result of TSL having set it to 1 by P_0 . Consequently, P_1 will busy wait, and not enter its critical section, which is the correct behavior.

Note that we must allow for a context switch to occur at anytime, and the main concern here is if a context switch were to occur within TSL, just after

```

shared int lock = 0;

P0           P1
while (! TSL(&lock));
< critical section >
lock = 0;          while (! TSL(&lock));
< critical section >
lock = 0;

```

Figure 5.9: A solution to mutual exclusion using TSL.

having recorded the old value of `lock`, and before setting `lock` equal to 1. But this cannot happen, because TSL runs atomically; it cannot be interrupted, and so no context could happen while it is running. This mimics the TSL instruction, which carries out the two operations atomically.

Notice how simple the entry code is in Figure 5.9. It is much simpler and that of Peterson's solution. And in fact, this code will enforce mutual exclusion (and more generally, satisfy all four requirements) not just for two processes, but for any number. And so we see the value of adding TSL to an instruction set.

Since the code in Figure 5.9 is so critical to performance, it is worth trying to improve it if possible. Note that a simple check as to whether the variable `lock` equals 1 is much cheaper than applying the TSL instruction to `lock`. Consequently, we could modify the code of Figure 5.9 to first do the simple test, is `lock` equal to 1, and if so, busy wait by rechecking using the low overhead test. If the test fails, we can then do the more expensive test involving TSL, which is necessary to atomically check the `lock` value and then that it be set to 1. This improved code, called *Test and Test-and-Set-Lock*,

is shown in Figure 5.10.

```
shared int lock = 0;

do {
    while (lock == 1);           // cheap
} while (! TSL(&lock));        // expensive
< critical section >
lock = 0;
```

Figure 5.10: Improved entry code using the Test and Test-and-Set-Lock code pattern.

We presented the code in Figure 5.9 (and Figure 5.10) for illustrative purposes, but in actuality, since there is no atomic TSL C procedure, the entry and exit code would be coded in assembly language, which is shown in Figure 5.11 (corresponding to the simpler code in Figure 5.9). The entry code consists of three instructions: the TSL instruction, which does the test-and-set-lock operation. We see that TSL actually takes two operands, a register REG, and a memory address, lock. TSL will load REG with the value contained at memory address lock, and store a 1 in lock, both operations done atomically. The next instruction compares the value just stored in the register to 0, checking whether it is equal to 0. If not equal to 0, the next instruction causes a looping back to the TSL instruction, resulting in a busy wait. This code pattern is known as a “*spin lock*.” The exit code is simple: it simply stores a 0 in lock.

There is still something bothersome about these solutions, both Peter-

Critical section entry code

```

; assume lock initially 0
loop: TSL REG, lock    ; atomically {load REG with lock
                        ;       and store 1 into lock}
        CMP REG, #0      ; is REG (was lock) equal to 0?
        JNE loop         ; if not equal to 0, check again
                        ; also known as a "spin lock"

```

Critical section exit code

```
MOV lock, #0      ; reset lock to 0
```

Figure 5.11: Entry and exit code using TSL in assembly language.

son's solution and the solutions that use the TSL instruction. The problem is that both involve busy waiting. Once it is determined that a process cannot proceed to enter critical section, why should the process continue repeatedly checking a variable that will not change until a context switch occurs? It will be good if the process simply gave up the CPU, and allow the other process to run without any wasting of CPU cycles.

This is addressed by the code shown in Figure 5.12. For the entry code (which is where the busy waiting may happen), we add a call to `yield` such that, if the `lock` is closed, rather than repeatedly checking and thus busy waiting, the process simply gives up with CPU. This seems to solve our problem! However, we will need to take a closer look at how `yield` is implemented, and we will see that the problem has not completely gone away. But for now, this is a substantial improvement.

Entry code

```
while (! TSL(&lock)) {           // lock closed
    yield();                      // give up CPU
}
```

Exit code

```
lock = 0;                         // open lock
```

Figure 5.12: Improved entry code by adding `yield`, to avoid busy waiting (for the most part – but is there any busy waiting in `yield`?).

While we've done our best to remove busy waiting, is busy waiting always bad (and, is there a price to pay for removing busy waiting)? In general, if the expected waiting time (amount of time a process has to wait before entering a critical section, which depends on the time spent by other processes in their critical section) is less than the scheduling overhead, then busy waiting is acceptable. Also, if we have lots of spare/idle CPUs, which can then be used to do the busy waiting and thus not interfere with the real work being done by the other CPU(s), then busy waiting is acceptable. Finally, there are situations where blocking, i.e., giving up the CPU, is not an acceptable option. A good example is during execution inside the kernel in certain situations. Consequently, our only option may be busy waiting.

To be more precise on how costly busy waiting is, consider the time spent in a critical section. The chances of a context switch occurring increase with the length of time spent in the critical section. If there is a context switch to a process seeking entry in their critical section, there will be busy waiting.

This will cause a waste of up to an entire quantum; consequently, this is the cost.

Given this, it is always worth trying to minimize the *time* spent in a critical section. This is not always possible, but if it is, it is worth doing. This is especially true for critical sections in kernel code, which are obligatory and affect all processes. Keep this in mind when we discuss the implementation of semaphores, below.

We have focused quite a bit on the problem of busy waiting in the entry code that protects critical sections. But there is another problem with the code patterns we have studied so far: they are somewhat unintuitive, and not so easy to understand. The code for Peterson solution is especially unintuitive (the n -process version is especially difficult to understand, but even the 2-process version is not trivial), and even the solution using TSL is not completely obvious. Fortunately there is a solution that addresses both of these problems.

5.8 Semaphores

A *semaphore* is a synchronization variable that has the following properties. It takes on integer values, and operating on it can cause a process to block or unblock. There are two, and only two, operations that can be applied to a semaphore, as shown in Figure 5.13.

The `wait` operation will decrement the integer value of the semaphore,

<code>wait(sem)</code>	Decrement the integer value of the semaphore <code>sem</code> , and if it is negative, block the calling process
<code>signal(sem)</code>	Increment the integer value of the semaphore <code>sem</code> , and if there are any processes blocked on <code>sem</code> , unblock one of them

Figure 5.13: Semaphore operations.

and if the value becomes negative, the process that called `wait` will block, causing a context switch to another process (that is in the READY state). The process that just blocked will be in the BLOCKED state, and so will not be chosen by the scheduler to run until it is changed to the READY state.

The `signal` operation will increment the value of the semaphore, and if there are any processes that had blocked in the past by having called `wait` (on the same semaphore), one of those processes will be unblocked, i.e., changed to the READY state.

Note that if there is more than one process that is blocked, it remains unspecified as to which of the multiple processes is unblocked, only that one of them will be unblocked. Many operating systems will unblock processes in FIFO (first-in-first-out) order, but that is an implementation feature, and not part of the specification that we can rely on as programmers.

Finally, we emphasize that no other operations are allowed; only `wait` and `signal` as specified are allowed. In particular, it is not permitted to *test* the value of a semaphore. Consequently, we should not view a semaphore as

a normal integer to which one can apply normal integer operations, such as arithmetic and checking equality or inequality to other integers. This is very important, and we will see why.

We can use semaphores to very effectively solve the mutual exclusion problem. In Figure 5.14, we see processes, P_0 and P_1 , with their critical sections, protected by very simple entry and exit code. The entry code consists of calling `wait(mutex)`, and the exit code consists of calling `signal(mutex)`, where `mutex` is a semaphore that has been initialized to 1. Semaphores are, by nature, shared variables, so all processes will be able to access them by name.

```
sem mutex = 1;           // declare and initialize

P0
  wait (mutex);
  < critical section >
  signal (mutex);

P1
  wait (mutex);
  < critical section >
  signal (mutex);
```

Figure 5.14: Using a semaphore to solve the mutual exclusion problem.

Let's say P_0 executes first. When it executes `wait(mutex)`, since `mutex` equals 1, the value of `mutex` becomes 0 and P_0 can proceed into the critical section. Say there's a context switch to P_1 . When P_1 executes `wait(mutex)`, since `mutex` equals 0, the value of `mutex` becomes -1, and since this is negative, and P_1 will block. This is the behavior we want; only one process in a critical section at a time.

Importantly, when P_1 blocks, *it gives up the CPU, allowing P_0 to resume.* In particular, there is no busy waiting by P_1 (this is not absolutely true, as we shall soon see, but to a large degree, it is true.) Consequently, this solution is much more efficient than the previous ones, where we had a lot of busy waiting.

Secondly, the solution is very intuitive. Let's interpret the value of the semaphore as follows: Upon calling `wait`, if the semaphore's value is 1 (or greater) it means “*go*,” and if its value is 0 (or less), it means “*stop*.” Upon calling `signal`, regardless of the semaphore's value, if there is a process waiting on that semaphore, it will be allowed to proceed; in other words, calling `signal` always means “*go*,” though not for the process that calls `signal`, but for a waiting process, *if* one exists. If there are multiple waiting processes, then the “*go*” applies to one, and only one, of the waiting processes. If there are no waiting processes, then there is no current process to which to apply “*go*,” *but it is remembered for any prospective call to wait.*

Applying this interpretation to the code in Figure 5.14, the semaphore `mutex` is initialized to 1, and so when P_0 is about to enter its critical section and calls `wait` on the semaphore `mutex`, it is a “*go*” and so P_0 can proceed into the critical section. However, since the integer value of `mutex` is 1, it will be decremented to 0, meaning “*stop*” for the *next* call to `wait`. And indeed, when P_1 calls `wait`, since the value of `mutex` is 0, P_1 will stop and not enter its critical section. When P_0 exits its critical section, it calls `signal` on `mutex`, and since there is a blocked process (on `mutex`), P_1 , it will be unblocked,

allowing it to “go;” the value of `mutex` will also be incremented.

This “stop-and-go” interpretation corresponds to the signaling used by trains, indicating when they must stop and go for a portion of a track that is shared. In fact, the term “semaphore” originates from this interpretation. Semaphores (for synchronizing processes) were invented by the famous computer scientist, Edsger W. Dijkstra, at the time a professor at the Eindhoven University of Technology in the Netherlands.

There is a form of semaphores, called *binary semaphores*, which are limited to taking on the values 0 and 1, and that fits the scenario and interpretation we have provided even better (because 0, and only 0, means “stop;”, and 1, and only 1, means “go;” there are no values greater than 1 or less than 0 to interpret, which we did not deal with in our example above but would in a more general situation). With binary semaphores, `wait` and `signal` are defined as shown in Figure 5.15.

<code>wait(bsem)</code>	If the value of the semaphore <code>bsem</code> is 1, decrement it to 0, otherwise block the process
<code>signal(bsem)</code>	If the value of the semaphore <code>bsem</code> is 0, increment it to 1, and if there are any processes blocked on <code>bsem</code> , unblock one of them

Figure 5.15: Binary semaphore operations.

Notice that our solution in Figure 5.14 works, regardless whether we use normal semaphores or binary semaphores. The difference is that normal

semaphores lend themselves to uses beyond mutual exclusion, as we will see in the next example.

But before leaving the example in Figure 5.14, let's note some features of the solution and summarize our observations. First, it correctly solves for mutual exclusion, and satisfies all four requirements. To see that this is true, we will have to look at how semaphores and their operations are implemented.

Second, it is very efficient. As soon as it is determined that a process cannot proceed, that process will block and give up the CPU, rather than spend significant time busy waiting. We will see that there is actually a little bit of busy waiting (when we see how semaphore operations are implemented), but the amount of time is very small compared to our previous solution.

Third, the solution is very simple and intuitive. It also works for any number of processes, not just two. And so the use of semaphores is very appealing.

Semaphores are not limited in their use to solving mutual exclusion. Consider the code in Figure 5.16. Let's see what it does. Let's assume that P_1 runs first. It executes the `wait` statement on semaphore `cond`, which was initialized 0. Consequently, P_1 will block. When P_0 runs, it will execute code that is labeled “to be done before P_1 ,” which means code that it is supposed to execute before P_1 is able to execute its code. When P_0 finishes executing this code, it executes the `signal` statement, causing P_1 to unblock. P_1 will now run, and execute the code that is labeled “to be done after P_0 ,” i.e., any

code that is supposed to execute after P_0 executes its code.

```
sem cond = 0;

P0                                P1
< to be done before P1 >      wait (cond);
signal (cond);                      < to be done after P0 >
```

Figure 5.16: Using semaphores to order the execution of processes.

Consequently, if we have two processes and we would like to fix the order in which they execute, we can do so using semaphores. In the example of Figure 5.16, we declared a semaphore `cond` and initialized it to 0. Then, by adding the `wait` and `signal` statements according to the pattern in Figure 5.16, we are able to force P_0 to execute its code before P_1 can execute its code, regardless of any context switching that takes place. Contrast this use of semaphores for ordering processes with their use in implementing mutual exclusion. And in fact, the usage of semaphores is not limited to just these two ways.

Before we go further, let's understand an important fact about what semaphores provide and what they don't provide. *Semaphores only provide synchronization.* Recall that, by synchronization, we mean when events in two processes are made to occur at the same time. With a single CPU, this amounts to forcing one process to wait when it reaches its event, to allow the other process to run so that it can reach its event, and at that point, we can say that the events are synchronized.

We've seen how semaphores are used to cause processes to wait. What is especially interesting is that, when processes use `wait` and `signal`, while this allows them to synchronize, *there is no information transfer that takes place*. In other words, by executing `wait` or `signal`, a process cannot learn anything about another process that may have called `wait` or `signal`!

You might say, can't a process look at the integer value of the semaphore, and see that if it changed, and if that process did not cause the change, then it must have been done by some other process, and so the process can learn something from this. However, a process is precluded from checking the value of the semaphore! It can only call `wait` or `signal`, and nothing else. There is a no operation that checks the value of the semaphore and allows it to be compared to another value so that it could be used in a conditional statement.

But, you might say, if a process blocks due to calling `wait`, can't it tell that it blocked? But how would it do so? Once a process unblocks due to another process having call `signal`, how will the unblocked process sense that it was blocked? You might say, can't a process read the time just before calling `wait`, and then read the time just after calling `wait`, and determine that if a lot of time went by, then it must have blocked? The answer is *no*. Perhaps the reason that so much time went by is that the scheduler simply did not allow the process to run. Even if the clock interrupt has gone off to allow the scheduler to make a decision as to which process to run next, it might simply choose another process.

So a process that calls `wait` cannot make any deduction as to whether it blocked or not. As this is the only way that processes affect each other – by causing each other to block or unblock – and since processes cannot tell whether they blocked or not, no information transfer is possible between processes.

This constraint of not being able to know the integer value of a semaphore may seem somewhat artificial. Why not allow processes to check these values? For example, imagine that a process is about to call `wait`, but before doing so, it would like to check the value of the semaphore to determine if it will block or not. If it sees that the value of the semaphore is 0 or less, it will know that it will block, and so might decide to do something else.

The fallacy with this reasoning is that the process cannot know what might happen between checking the value and executing `wait`. Perhaps it checks the value of the semaphore, sees that the value is 1, and then calls `wait` expecting not to block. But, if there is a context switch just after checking, and another process runs that causes the semaphore value to change to 0, by the time the original process resumes, executing `wait` will cause it to block! And there is no way to solve this. If it were to check again, that still would leave it open to another context switch immediately after checking, and so it cannot rely on the value it observed. For this reason, checking the value of a semaphore is not made possible, because it serves no purpose and can only be misused.

5.9 Implementing Semaphores

Let's now look at how semaphores are implemented. Figure 5.17 shows one possible implementation, which happens to follows the specification very closely.

Semaphore s = [n, L]

n: takes on integer values, negative OK

L: list of processes blocked on s

Operations

```
wait (sem s) {
    s.n = s.n - 1;
    if (s.n < 0) add calling process to s.L and block; }

signal (sem s) {
    s.n = s.n + 1;
    if (s.L !empty) remove/unblock a process from s.L; }
```

Figure 5.17: An implementation of semaphores.

We implement the data structure for a semaphore as an integer value n, and a list of potentially blocked processes L. The `wait` operation takes a semaphore s, decrements its integer value `s.n`, and if the value is negative, adds the calling process to the semaphore's list of blocked processes `s.L`, and blocks itself (which essentially changes the calling process's state to BLOCKED, and calls a form of `yield` causing the process to give up the CPU without necessarily indicating which process to yield to, but rather allowing the scheduler to select a ready process to run).

The `signal` operation takes a semaphore `s`, increments the integer value `s.n`, and if the semaphore's list of blocked processes `s.L` is not empty, removes one and unblocks it (which essentially changes the chosen process's state to READY). Note that, when unblocking a process, this does not mean the process will immediately run. Rather, it only means that when the scheduler runs at some point in the future, the unblocked process, now in the READY state, is eligible for being chosen to run.

Furthermore, a particular implementation will determine the order in which processes are unblocked, which is typically FIFO, but does not have to be. Regardless of the unblocking order, the programmer would not be able to take advantage of the unblocking order as it is not part of the semaphore interface specification. Rather, it is only an implementation detail, which can change from one operating system to another.

The code shown in Figure 5.17 is not the only possible implementation. Consider the implementation in Figure 5.18. The `wait` operation first checks to see whether the integer value of the semaphore is less than or equal to 0, and if so, the calling process is added to the semaphore's list of blocked processes, and the process blocks. If the integer value of the semaphore was not less than or equal to 0, i.e., it is positive, then the integer value is decremented and `wait` is done.

The `signal` operation first checks whether its list of blocked processes is not empty, and if there are blocked processes, removes one and unblocks it. Otherwise (if the list of block processes is empty), the value of the semaphore

Semaphore $s = [n, L]$

n : takes on integer values, non-negative

L : list of processes blocked on s

Operations

```

wait (sem s) {
    if (s.n == 0) add calling process to s.L and block;
    else s.n = s.n - 1; }

signal (sem s) {
    if (s.L !empty) remove/unblock a process from s.L;
    else s.n = s.n + 1; }

```

Figure 5.18: An alternative implementation of semaphores.

is incremented and **signal** is done.

These two implementations are equivalent in that the **wait** and **signal** operations will have the same *functional* effects, even though they accomplish them in different ways. You should convince yourself that they are indeed “equivalent,” in this sense.

We now come to a very important point regarding the implementation of semaphores. The procedures **wait** and **signal** *must be atomic*. Consequently, the bodies of **wait** and **signal** are themselves critical sections! Note that we are now discussing different critical sections than the ones that may be in the programmer’s code, as discussed above. The bodies of **wait** and **signal** are critical sections *as they relate to each other*, but are *independent* of other critical sections in the programmer’s code.

Consequently, we require mutual exclusion for these critical sections. This seems somewhat circular: we use the semaphore `wait` and `signal` operations to enforce mutual exclusion for critical sections, but the bodies of the `wait` and `signal` operations themselves require mutual exclusion. The idea is to somehow implement mutual exclusion for the semaphore operations, and then, we can use the semaphore operations `wait` and `signal` as the entry and exit code to protect other critical sections.

Why would we do this? Because once we have the semaphore abstraction, the programmer can use this to protect critical sections, to order the execution of processes, etc. Once we have semaphores, it is very easy to use them for these purposes. But we are left with the problem, how do we make `wait` and `signal` atomic? We certainly cannot use semaphores to implement semaphores! Do we have any other way of implementing mutual exclusion?

Yes, in fact, we have two ways: Peterson's solution, or using TSL! If our machine includes the TSL instruction, we will use the method shown in Figure 5.9 (or its performance-improved variants). Otherwise, we will use Peterson's solution, as given in Figure 5.7. Both work, and so we can achieve atomicity for the semaphore `wait` and `signal` operations.

But, doesn't this mean that there will be busy waiting, and if so, doesn't this go against one of the reasons for using semaphores, to avoid busy waiting? To the first part of the question, the answer is a definitive *yes, there will be busy waiting*. But, it's not as bad as we might think. The busy waiting is going on *at a lower level*, specifically at the level of the code for `wait` and

`signal`. At that level, the amount of busy waiting at that level will actually be quite small.

It is worth reviewing how and when busy waiting happens in Peterson's solution and in the TSL solution. It happens if a process is trying to enter its critical section, while another process is already in its critical section. But the critical sections that we are talking about here are the bodies of `wait` and `signal`, and the amount of code in these bodies is very small. This means that a process will be executing in these critical sections of code only momentarily. Thus, the chance of a context switch occurring during these brief periods is also very small.

It is only when a context switch occurs that will cause a process (that is now given the CPU) to busy wait. If it does have to busy wait, then it will have to busy wait for the rest of the quantum, until it is preempted, like before. But since the *likelihood* of a context switch occurring during a process's execution in the critical sections of `wait` and `signal` bodies is very small, the overall amount of busy waiting will ultimately be small.

Benjamin Franklin said, "nothing can be said to be certain except death and taxes." Well, we can add busy waiting to the list; it is certain that, at some level in an operating system (and indeed, even in the lower level hardware), there will be busy waiting! It might not be a lot, but we can never get rid of it completely.

5.10 Summary

Synchronization is the arranging of events to occur at the same time. Process synchronization is when the events belong to processes, and is achieved by having one process, which has reached its event, to wait for the other process until it reaches its event.

The Credit/Debit problem illustrates the problem of race conditions. To avoid race conditions, we must identify critical sections, which are to run atomically relative to each other. This requires mutual exclusion. A good solution for mutual exclusion must satisfy four requirements. Two solutions that satisfy these requirements are Peterson's solution, which can be completely implemented in software, or a lock based on using the TSL instruction, which must be available in hardware.

Mutual exclusion can also be solved using semaphores, which are more intuitive to use than the other solutions, and avoid most of the busy waiting in those solutions. In fact, semaphores can be used for other purposes, such as ordering the execution of processes. However, semaphore operations themselves require mutual exclusion, which can be implemented using the other methods. While this means that busy waiting is reintroduced, it occurs much less frequently. Despite this, we see that busy waiting can never be completely removed.

5.11 Exercises

1. What is the general (dictionary) meaning of the word “synchronize”?
2. What does “process synchronization” mean?
3. How are the general word “synchronization” and the term “process synchronization” related?*
4. What are the uses of synchronization?
5. What is a race condition?*
6. In the Credit/Debit problem, how is it possible for the resulting balance to be \$1100?*
7. How is it possible for the resulting balance to be \$990?*
8. Are there any other resulting balances possible?**
9. In the code on Figure 5.1, where is the race condition?*
10. What is a critical section?
11. What does “run atomically with respect to each other” mean?
12. How are critical sections used to avoid race conditions?*
13. What if we had two processes, and simply made their entire code bodies critical sections, would there be any race conditions?**

14. What is wrong with simply making the entire code body a critical section?***
15. What does “mutual exclusion” mean?
16. What is the general way of achieving mutual exclusion?
17. What should the `<entry code>` do?
18. What should the `<exit code>` do?
19. In the code in Figure 5.3, is the `<entry code>` part of the kernel?**
What about the '`<exit code>`?** What about the `<critical section>`?**
20. What are the requirements for a good solution to achieve mutual exclusion?
21. What requirement is the most fundamental one, and why?**
22. Which requirements are needed to prevent trivial solutions that are of no real value?**
23. Which requirement removes any dependence on the machine’s performance?**
24. What is wrong with the “Software Lock” solution? Which requirement is not met?*
25. What is wrong with the “Take Turns” solution? Which requirement is not met?*

26. What is wrong with the “State Intention” solution? Which requirement is not met?*
27. What is wrong with the “Disabling Interrupts” solution? Which requirement is not met?*
28. Does Peterson’s Solution work, i.e., are all the requirements met?
29. What is the problem with Peterson’s solution?*
30. What does the **TSL** instruction do?*
31. How can the **TSL** instruction be used to solve mutual exclusion?*
32. What are the pros and cons of this solution?*
33. What is a semaphore?
34. What does the **wait** operation do?
35. What does the **signal** operation do?
36. Is it possible to inspect the value of the semaphore, why or why not?*
37. How can semaphores be used to solve mutual exclusion?*
38. What are the pros and cons of this solution?*
39. What is meant by “busy-waiting”?*
40. How can semaphores be used to order the execution of processes?*

41. What is meant by “conditional synchronization”?**
42. What is meant by “pure synchronization”?*
43. Can semaphores (and nothing else) be used to transfer information, why or why not?***
44. How does the semaphore implementation in Figure 5.17 satisfy the semaphore specification in Figure 5.13?**
45. How does the semaphore implementation in Figure 5.18 satisfy the semaphore specification in Figure 5.13?**
46. Do the implementations in Figure 5.17 and Figure 5.18 have the same functional effect?**
47. Why must `wait` and `signal` be atomic?**
48. What does it mean that the bodies of `wait` and `signal` still need a mechanism for mutual exclusion?***
49. What mechanisms can be used for this purpose, and why?**
50. When using these mechanisms, why is it that busy-waiting still exists in `wait` and `signal` – can you provide an example/scenario?***
51. If busy-waiting still exists, why even use `wait` and `signal`?***
52. Consider protecting a critical section using semaphores vs. using Peterson’s solution: which has more busy waiting and why?***

53. Are the semaphore implementations in Figure 5.19 equivalent, why or why not?***

Implementation 1

```
wait (sem s) {
    s.n = s.n - 1;
    if s.n < 0 {
        addProc (me, s.L);
        block (me);
    }
}

signal (sem s) {
    s.n = s.n + 1;
    if (! empty (s.L)) {
        p = removeProc (s.L);
        unblock (p);
    }
}
```

Implementation 2

```
wait (sem s) {
    if s.n ≤ 0 {
        addProc (me, s.L);
        block (me);
    }
    s.n = s.n - 1;
}

signal (sem s) { // same
    s.n = s.n + 1;
    if (! empty (s.L)) {
        p = removeProc (s.L);
        unblock (p);
    }
}
```

Figure 5.19: Two implementations of semaphore wait and signal operations.

54. Do either of the implementations have a race condition, and if so, where?***
55. Edsger Dijkstra, the inventor of semaphores, describes a problem called “The Dining Philosophers,” where five philosophers sit around a table, thinking and dining, each with a plate of spaghetti (see Figure 5.20). The spaghetti must be eaten with two forks (Dijkstra was clearly not Italian!). There are a total of five forks, one positioned between each philosopher such that each philosopher has one fork to the left and one to the right.

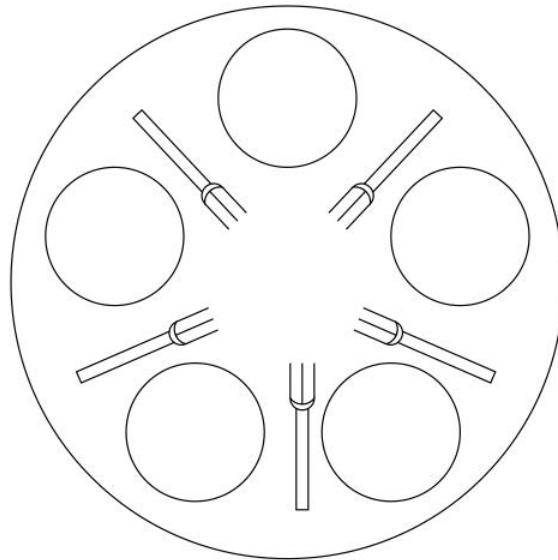


Figure 5.20: Five dining philosophers sit around a table, with plates of spaghetti and forks between them.

To eat, a philosopher must pick up both forks, the one to the left and the one to the right. If a philosopher is able to pick up only one fork, and the other one is currently being used, that philosopher cannot eat until the other fork becomes available. Each philosopher independently alternates between thinking (“philosophizing”) and eating.

Figure 5.21 shows skeletal code for a procedure, `doPhilosopher(i)`, to be called by philosopher identified by their integer ID, numbered 0 to 4. The rules for eating are as follows:

- a philosopher will proceed to eat only if both forks are picked up
- a fork will not be used by two philosophers at the same time

```
DoPhilosopher (int i) {
    while (TRUE) {
        Think ();
        PickupFork (i);
        PickupFork ((i+1)%5);
        Eat ();
        PutdownFork ((i+1)%5);
        PutdownFork (i);
    }
}
```

Figure 5.21: Procedure called by each of the dining philosophers.

In the Dining Philosopher's Problem, why is mutual exclusion needed?*

56. In Figure 5.21, can you identify the critical section(s)?*
57. How can mutual exclusion be achieved?*
58. How can starvation be avoided (i.e., eventually, a philosopher will be able to eat, rather than waiting forever)?*
59. Using semaphores and shared variables, modify the code so that the rules for eating are followed.**
60. Given your answer to how starvation can be avoided, modify your solution appropriately.***
61. Generalize your solution so that it will work for $n > 1$ philosophers.***

62. In the “Readers/Writers Problem,” there are one or more “reader” processes and one or more “writer” processes, and a common file they are all trying to access. A reader will only read the file, while a writer can also write the file, and all readers and writers must abide by the following rules:
- If there are only readers, they must all be simultaneously allowed to read the file
 - If a writer is writing the file, there can be no readers or other writers simultaneously allowed to access (read or write) the file
 - If there are only readers, once a writer appears, it must wait for those existing readers (but not new ones) to complete before it can begin writing
 - If there is a writer, all new readers and writers must wait until the writer is done
 - If there are multiple readers and writers waiting, the writers take precedence

Figure 5.22 shows skeletal code for `doReader()` and `doWriter()`.

How are readers and writers treated differently in the Readers-Writers Problem, and why?**

63. Use shared variables and semaphores to solve the Readers/Writers Problem so that all the rules are followed.***

```
doReader ()
{
    < open file for reading >
    < read the file >
    < close file >
}

doWriter ()
{
    < open file for writing >
    < write the file >
    < close file >
}
```

Figure 5.22: Procedures called readers and writers for the Readers/Writers Problem.

64. Modify your solution to the Readers-Writers Problem?*** so that it avoids starvation (so no reader, nor no writer, may have to wait forever).***

Chapter 6

Interprocess Communication

We now consider how to support programs that consist of a set of *cooperating* processes, i.e., processes that coordinate their activities by communicating with each other. Why would a programmer want to structure a program as a set of cooperating processes?

One reason is *performance*. Multiple processes allow the program to exploit any inherent parallelism in the computation. It also allows some parts to proceed, while others may be blocked, e.g., waiting for I/O.

The second reason is *modularity*, encouraging the development of reusable self-contained programs. Each such program may do some useful task on its own. Each may also be useful as a subtask within a larger program. Consequently, the small programs can run as processes that then communicate as components of a larger computation.

6.1 Cooperating Process Structures

There are numerous examples of how we might structure cooperating processes. One example is the *pipeline*, as shown in Figure 6.1. In this example, there are three processes, P_1 , P_2 , and P_3 , which are connected via communication channels such that the output of P_1 forms the input of P_2 , and the output of P_2 forms the input of P_3 . All three processes can proceed in parallel, as long as they have work to do.

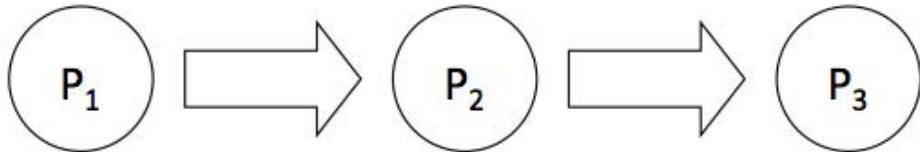


Figure 6.1: Three processes connected according to a pipeline.

The second example is shown in Figure 6.2. Here we have the *client/server model*, which is an especially useful way of organizing multiple-process programs where the processes are distributed over a network. There are two processes, the client and server. The *client* process is generally a short-lived process, which acts on behalf of a user (and typically runs on the user's computer, e.g., a laptop), such as an email client or web browser. The *server* process is generally a long-lived process that runs on a server machine. It simply waits for requests from a client, any client, generates a result, and sends it back to that client. A good example is a web server, where web browser clients make requests for web pages, sending the requests to the ap-

ropriate servers that can satisfy these requests, and the servers send back the results.

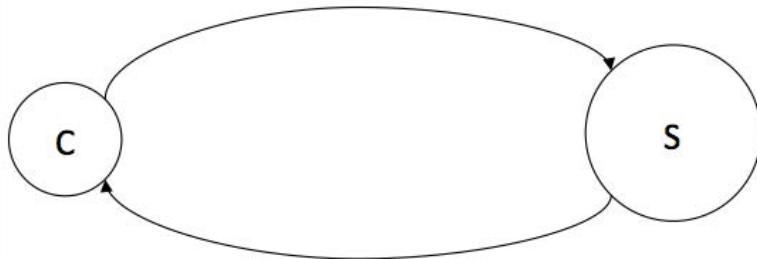


Figure 6.2: The client/server model.

The third example is shown in Figure 6.3. This is a hierarchical structure, where a parent process generates multiple child processes to carry out sub computations. The parent then waits for the results, and when the child processes are done, the parent can aggregate the results. In this way, the child processes can each operate in parallel, and if there are multiple CPUs, the computation can be done quickly. But even if there is only a single CPU, there is a benefit in that whenever a child process blocks to wait for I/O, for example, another can make progress. This would not be possible if there were only a single process to do the entire computation (unless it was multi-threaded – but if so, then we could apply the same idea to a set of threads structured in a hierarchy. In this chapter, we will limit ourselves to cooperating processes that are single threaded, knowing that most of the concepts are just as applicable to cooperating threads that are within a process or are part of multiple processes. Consequently, you may safely replace the work

“process” with “thread” throughout this chapter).

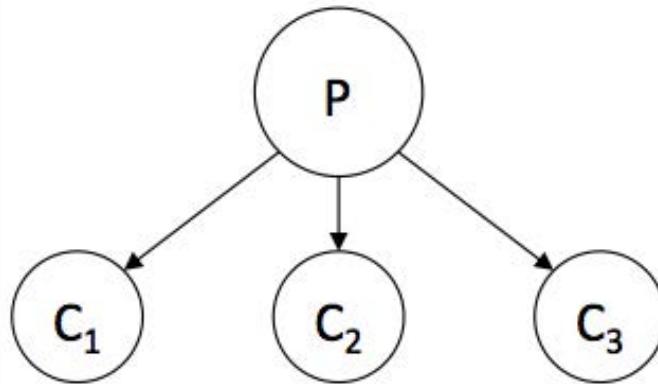


Figure 6.3: Cooperating processes organized as a parent/child hierarchy.

6.2 Inter-Process Communication

For this cooperation between processes to take place, the processes need to be able to communicate. The operating system provides this facility, which is called *inter-process communication* (IPC).

IPC requires two distinct mechanisms: one for *data transfer*, and one for *synchronization*. Importantly, any form of IPC requires both; if only one is provided, the IPC facility is not complete. We saw in the last chapter that semaphores provide synchronization, but only synchronization. Consequently, semaphores by themselves are not adequate for IPC. We need more, specifically, a data transfer mechanism.

In this chapter, we will consider three abstractions for IPC. The first is

the combination of shared memory and semaphores. The second is monitors. The third is message passing. Each one is different in how the programmer can express communication between processes; some are better for certain situations than others. But, all of them must support both data transfer and synchronization, and for each one, it is important that we clearly identify the form of those mechanisms, how they are achieved, and how they work.

6.3 The Producer/Consumer Problem

We will present these IPC abstractions through their use in solving a classical problem, called the “*Producer/Consumer problem*,” as shown in Figure 6.4. Here we have two processes, a process that produces items, called the “Producer,” and a process that consumes items, called the “Consumer.” In this example, the Producer continually produces or generates integers. The Consumer continually consumes or processes the integers generated by the Producer.

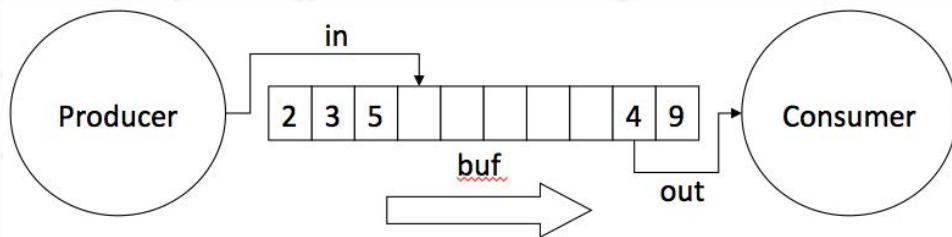


Figure 6.4: The Producer/Consumer problem.

To support the transfer of these integers from the Producer to the Con-

sumer, we use a circular buffer. A *buffer* is simply a temporary storage area to hold items; in this example, the items are the integers that are in transit between the Producer and the Consumer. After an integer has been produced, but before it has been consumed, it needs to be placed in some temporary holding area, and this is precisely the purpose of a buffer.

The buffer is shown as an array called “`buf`,” where sequential entries of `buf` store sequentially produced integers by the Producer. A variable, called “`in`,” always points to the next available slot in `buf`. When the Producer generates a new integer, it is placed in the slot pointed to by `in`, and `in` is incremented to the next slot (for now, assume that that slot is empty and thus available).

Another variable, called “`out`,” always points to the slot containing the oldest integer that has yet to be removed. When the Consumer is to remove an integer, it obtains it from the slot pointed to by `out` (for now, assume that that slot is filled with an integer), and `out` is incremented.

Of course, the array `buf` is of finite length. Eventually, both variables `in` and `out` will reach the end of `buf`. When this happens to either of them, the variables are reset to point to the first element of the array, effectively organizing the array in a circle where the next element after the last slot is the first slot. This is why we call `buf` a “circular buffer.”

Through the use of the circular buffer, the Producer and the Consumer can communicate, and thus, cooperate, allowing the Producer to feed the Consumer. Given that the Producer and the Consumer are separate pro-

cesses, with their own separate memories, we have to identify the memory for the circular buffer: Where is this memory? Secondly, if the buffer is empty, we want the Consumer to wait until something becomes available: How is this done? Similarly, if the buffer is full, we want the Producer to wait until a slot becomes available: How is this done?

Consider the solution to the Producer/Consumer problem shown in Figure 6.5. First, we see there are various shared variables. We have a buffer array of integers called “buf.” We also have two integers called “in,” and ‘out,’ that represent pointers into the buffer array. The array will be managed as a circular buffer as described earlier.

```
shared int buf[N], in = 0, out = 0;
Producer                                Consumer
while (TRUE) {                           while (TRUE) {
    buf[in] = Produce ();           Consume (buf[out]);
    in = (in + 1)%N;              out = (out + 1)%N;
}                                         }
```

Figure 6.5: A possible solution to the Producer/Consumer problem.

There are also two programs, one for the producer process, called “Producer,” and one for the consumer process, called “Consumer.” The Producer consists of a `while` loop, in which its first statement calls `Produce`, which generates and returns an integer, which is then placed in `buf` in the slot pointed to by the variable `in`. The variable `in` is then incremented, and then taken modulo `N` so that `in` will properly wrap around the circular buffer.

The Consumer program also consists of a `while` loop, in which its first statement calls `Consume`. `Consume` takes as a parameter the integer located in the slot of `buf` pointed to by the variable `out`. This is the item it will consume, or process. The variable `out` is then incremented, and then taken modulo `N` so that our will properly wrap around the circular buffer.

This code will not work properly in that there is no synchronization between the processes. In general, the Consumer should wait for an integer to be produced before it consumes it. But here, the Consumer simply assumes that an integer is present in the buffer for consuming. Consequently, this requires synchronization, i.e., having the Consumer wait until the Producer makes an integer available.

What about the Producer? It also requires synchronization. If at some point the buffer becomes full, the Producer must wait until a slot becomes available, and this will only happen after the Consumer has had a chance to remove an item.

So we see that, while data can be transferred between processes, achieved via the shared memory, there is no synchronization, which will lead to problems. We need a synchronization mechanism, and once we have that, combined with the data transfer mechanism of shared memory, we have IPC.

Fortunately, we've already encountered a perfectly useful synchronization mechanism in the form of semaphores. We can use semaphores to provide synchronization, and thus, fix our solution to the Producer/Consumer problem.

6.4 Semaphores + Shared Variables

Recall that a semaphore is a synchronization variable that takes on integer values, and may have an associated list of processes that are waiting for the semaphore to be signaled. The use of semaphores is supported by two operations, `wait` and `signal`, as was shown in Figure 5.13 in Chapter 5. How might we use `wait` and `signal` to add the proper synchronization to solve the Producer/Consumer problem?

Figure 6.6 shows this solution. We declare two semaphores, called `filledslots` and `emptyslots`. A good convention in naming semaphores is that they describe the condition that is being waited for or being signaled. For example, `wait(emptyslots)` signifies that a process is waiting for the availability of empty slots, and `signal(emptyslots)` signifies that empty slots are now available. Similarly, `wait(filledslots)` signifies that a process is waiting for the availability of filled slots, and `signal(filledslots)` signifies that filled slots are now available.

Notice how the semaphores are initialized. The semaphore `filledslots` is initialized to 0, because there are initially no filled slots. The semaphore `emptyslots` is initialized to N (where N is the number of slots in the buffer), because there are initially N empty slots. When we discussed semaphores in the previous chapter, we saw two uses, one for mutual exclusion and the other for ordering processes. Here we see a new use, that of *counting* (semaphores used in this way are commonly called “counting semaphores”).

```

shared int buf[N], in = 0, out = 0;
sem filledslots = 0, emptyslots = N;

Producer                                Consumer
while (TRUE) {                            while (TRUE) {
    wait (emptyslots);                  wait (filledslots);
    buf[in] = Produce ();              Consume (buf[out]);
    in = (in + 1)%N;                  out = (out + 1)%N;
    signal (filledslots);             signal (emptyslots);
}
}

```

Figure 6.6: A solution to the Producer/Consumer problem using semaphores and shared memory.

Let's see how the code in Figure 6.6 works. When the Producer begins executing, it enters the `while` loop, and calls `wait(emptyslots)`. Since `emptyslots` was initialized to `N`, the semaphore will be decremented, and since it is not negative, the Producer can proceed.

The Producer then goes ahead and fills the next buffer slot, and calls `signal(filledslots)` to indicate that there is a new filled slot. In fact, `filledslots` will be incremented, and if the Consumer is blocked, waiting for a filled slot, the Consumer will be unblocked.

The Producer will now repeat. If the Producer carries out this loop `N` times, prior to the Consumer running, it will have filled all `N` slots. At this point, `wait(emptyslots)` will have been called `N` times, resulting in `emptyslots` being equal to 0. Upon the next attempt to produce an integer, the Producer will block upon calling `wait(emptyslots)`, as there are no empty slots available, precisely the synchronization behavior we want.

Let's now turn our attention to the Consumer. When the Consumer begins executing, it enters the `while` loop, and calls `wait(filledslots)`. Since `filledslots` was initialized to 0, the semaphore will be decremented, and since it is negative, the Consumer will immediately block.

This is exactly as it should be. Since there are no filled slots, the Consumer should not be removing anything from the buffer. We want it to wait until the Producer has had a chance to fill a slot. When the Producer executes its `while` loop even just once, the Producer will call `signal(filledslots)`, waking up the Consumer.

The Consumer can then go ahead, obtain the item and empty the slot (by incrementing `out`), and call `signal(emptyslots)` to indicate that there is a new empty slot. In fact, `emptyslots` will be incremented, and if the Producer happens to be blocked, waiting for an empty slot, the Producer will be unblocked.

The Consumer will now repeat. If the Producer has filled all the slots such that it then blocks, the Consumer will be able to carry out this loop up to N times prior to the Producer resuming, and it will have emptied all N slots. At this point, `wait(filledslots)` will have been called N times, resulting in `filledslots` being equal to 0. Upon the next attempt to consume an integer, the Consumer will block upon calling `wait(filledslots)`, as there are no filled slots available, precisely the synchronization behavior we want by the Consumer.

While we looked at the extremes of the Producer having filled all slots

before the Consumer runs, or the Consumer having emptied all slots before the Producer runs, the program will work properly for intermediate scenarios, e.g., the Producer partially filling the buffer, the Consumer consuming only a portion of those, then the Producer filling up some more, etc. You should convince yourself that the code works by allowing context switches at a variety of times.

Note again the specific way that semaphores are being used here. They are being used to *count* the number of filled slots and the number of empty slots, and allowing the processes to synchronize with the special events of there being no filled slots and no empty slots. The Producer will wait if there are no empty slots, and the Consumer will wait if there are no filled slots. Furthermore, the calls to `signal` will allow the Consumer to unblock whenever there is a newly filled slot, and the Producer to unblock whenever there is a newly available empty slot. We might call this use of semaphores “general synchronization,” in contrast to the use we saw in the last chapter to achieve mutual exclusion.

It is interesting that mutual exclusion is not required. This seems odd given that there is a shared data variable used by both the Producer and the Consumer, namely the buffer `buf`, and one can imagine a race condition in the updating of the buffer by the processes. In fact, we do require mutual exclusion so that the same slot is not updated “at the same time” with indeterminate results. But mutual exclusion is achieved indirectly as a result of having the synchronization added so far. Next, we see a situation where

we must address mutual exclusion more directly.

Imagine a situation where we had multiple producers, each inserting integers they produce into the buffer as they are produced. This is shown Figure 6.7, where we have two producers, Producer1 and Producer2, both of which run a copy of the same code that was shown earlier for the Producer. Will this work correctly, even with the synchronization statements that we included that prevent adding to a full buffer and extracting from an empty buffer?

```

shared int buf[N], in = 0, out = 0;
sem filledslots = 0, emptyslots = N;

Producer1           Producer2           Consumer
while (TRUE) {          while (TRUE) {          while (TRUE) {
    wait (emptyslots);    wait (emptyslots);    wait (filledslots);
    buf[in] = Produce (); buf[in] = Produce (); Consume (buf[out]);
    in = (in + 1)%N;      in = (in + 1)%N;      out = (out + 1)%N;
    signal (filledslots); signal (filledslots); signal (emptyslots);
}                      }                      }

```

Figure 6.7: the Producer/Consumer problem, with two producers.

In fact, we now have a potential race condition between the two producer processes. Say Producer1 has just generated an integer and placed it in the buffer `buf` at the slot pointed to by `in`, and that there is a context switch to Producer2. Producer2 now generates an integer, and places it in `buf` at the same slot, because `in` has not been updated. This will overwrite the integer added by Producer1, which is effectively lost.

We see that the portions of the bodies of Producer1 and Producer2 are

critical sections, which require protection. The critical sections include the statement that calls `Produce` and adds the returned integer to `buf`, and the statement that updates `in`. These must run atomically.

We know how to achieve mutual exclusion through the use of a “*mutex*,” or mutual exclusion, semaphore. Consequently, we can add an additional mutex semaphore to our program, initialize it to 1, and then surround the critical sections with `wait(mutex)` and `signal(mutex)` statements. Indeed, this will also work if we had multiple consumers, to protect the critical section that includes consuming the integer located at `buf[out]` and then updating the variable `out`. This results in the code shown in Figure 6.8.

```

shared int buf[N], in = 0, out = 0;
sem filledslots = 0, emptyslots = N, mutex = 1;

Producer1, 2, ...
while (TRUE) {
    wait (emptyslots);
    wait (mutex);
    buf[in] = Produce ();
    in = (in + 1)%N;
    signal (mutex);
    signal (filledslots);
}

Consumer1, 2, ...
while (TRUE) {
    wait (filledslots);
    wait (mutex);
    Consume (buf[out]);
    out = (out + 1)%N;
    signal (mutex);
    signal (emptyslots);
}

```

Figure 6.8: Adding mutual exclusion when there are multiple producers and multiple consumers.

This solution will now work, though it is overly restrictive. Using the mutex semaphore to obtain mutual exclusion such that execution in any of the producers’ critical sections will now exclude execution in any of the

consumers' critical sections, and vice versa, is not necessary, and reduces potential parallelism. More generally, we can say that the group of critical sections for the producers and the group for the consumers do not require mutual exclusion *between* groups, only *within* each group. Consequently, the best solution is to have two mutex variables, one used by the producers (`pmutex`), and the other used by the consumers (`cmutex`), so that mutual exclusion obtained for the producers, and mutual exclusion obtained for the consumers, are kept separate. This is shown in Figure 6.9.

```

shared int buf[N], in = 0, out = 0;
sem filledslots = 0, emptyslots = N, pmutex=1, cmutex=1;

Producer1, 2, ...
while (TRUE) {
    wait (emptyslots);
    wait (pmutex);
    buf[in] = Produce ();
    in = (in + 1)%N;
    signal (pmutex);
    signal (filledslots);
}

Consumer1, 2, ...
while (TRUE) {
    wait (filledslots);
    wait (cmutex);
    Consume (buf[out]);
    out = (out + 1)%N;
    signal (cmutex);
    signal (emptyslots);
}

```

Figure 6.9: Giving the producers and consumers their own mutex semaphores, since their critical sections only require mutual exclusion within their own groups (the producers, or the consumers), and not between the groups.

While our solution Figure 6.9 works, and works well, one has to admit that it is not very easy to understand. In fact, it is easy to make a mistake in its coding, and even seemingly small mistakes can lead to major failures. For example, if the `wait` statements were interchanged (`wait(pmutex)` be-

fore `wait(emptyslots)`, or `wait(cmutex)` before `wait(filledslots)`), the processes may deadlock.

A *deadlock* occurs when processes are mutually waiting for each other, and because none can make progress to unblock each other, they will all remain blocked forever. And so while semaphores plus shared memory provide all the mechanism required for IPC, their use requires skill and programs based on their use are fragile. It would be nice to have, from a programmer's point of view, an easier-to-use form of IPC.

6.5 Monitors

A *monitor* is a programming language construct for IPC. Its elements include *variables*, which are effectively shared by processes and require controlled access, *procedures* that are used to access the variables and that provide mutual exclusion, and *condition variables* that are used for general synchronization.

Condition variables are used by passing them as parameters to `wait` and `signal` procedures, similar in usage to `wait` and `signal` for semaphores, but different in the way they work, as shown in Figure 6.10.

One more thing we need to know before we look at an example: *only one process can be active inside a monitor at any point in time*, which we call the “monitor rule.” By active, we mean running or able to run. All other processes must wait, and thus will block, upon trying to enter (i.e., when a monitor procedure is called). Only when the active process exits the monitor,

<code>wait(cond)</code>	Causes a process to block and wait for the condition <code>cond</code> .
<code>signal(cond)</code>	Check if there are any processes waiting for the condition <code>cond</code> , and if so, unblock one of them; if there are no such processes, do nothing.

Figure 6.10: Monitor synchronization operations.

or calls `wait` (making it temporarily inactive), will a process that is blocked, that was trying to enter the monitor, unblock and enter (any other processes blocked for this reason must continue waiting, according to the monitor rule).

Figure 6.11 shows the Producer/Consumer problem solved using a monitor. Everything inside the gray area is part of the monitor. First, we see that the code construct is labeled with the keyword, “monitor,” since it is part of the programming language, and has the name `ProducerConsumer`. In the body of the monitor, we see that there is a number of variables declared. These are variables that could be used throughout the monitor, but only inside the monitor. They are protected in the sense that they can only be accessed and modified by the code within the monitor. They are shared in the sense that, any process can access them, but only via the monitor procedures.

In our example, the variables include the buffer array `buf`; the integer variables `in` and `out` that point to the first empty slot and first filled slot, respectively; and a `count` variable, which will keep track of how many slots

```

monitor ProducerConsumer {
    int buf[N], in = 0, out = 0, count = 0;
    cond slotavail, itemavail;
    PutItem (int item) {
        if (count == N)
            wait (slotavail);
        buf[in] = item;
        in = (in + 1)%N;
        count++;
        signal (itemavail);
    }
    GetItem () {
        int item;
        if (count == 0)
            wait (itemavail);
        item = buf[out];
        out = (out + 1)%N;
        count--;
        signal (slotavail);
        return (item);
    }
}

Producer
while (TRUE) {
    PutItem (Produce ());
}

Consumer
while (TRUE) {
    Consume (GetItem ());
}

```

Figure 6.11: The Producer/Consumer problem solved using a monitor. The code in the gray area corresponds to monitor code.

are being used in buffer. Note that we did not need to maintain such a count in our solution when using semaphores, whereas here we do. In addition, there are two condition variables: `slotavail` and `itemavail`. Notice that they are not initialized to any values. This is because they actually do not store any values, as they effectively have no memory, as we will see.

The monitor includes two procedures: `PutItem` and `GetItem`. Let's look at each one. `PutItem` takes an integer as a parameter, and will store that integer in the buffer but at the next free slot indicated by the integer variable `in`. It first checks whether `count` equals `N` (the size of the buffer), which is to determine whether the buffer is full. If so, it calls `wait(slotavail)`, which causes the calling process to block on the condition `slotavail` (meaning,

“wait until a slot becomes available”).

If the buffer is not full, it continues by adding the passed integer to the buffer, just as in the solution we saw earlier when using semaphores, incrementing `count` because a new item was added to the buffer, and calling `signal(itemavail)`. If any processes happen to be waiting on the condition `itemavail`, one of them would be unblocked. If there is more than one process blocked, it is unspecified which one will be unblocked.

`GetItem` works as follows. It first checks whether `count` equals zero, which is to determine whether the buffer is empty. If it is empty, `wait(itemavail)` is called, which causes the calling process to block on the condition `itemavail` (meaning, “wait until an item becomes available”).

If the buffer is not empty, the process continues by removing an item from the buffer, again just as in the solution we saw earlier with semaphores, decrementing `count` because an item was removed from the buffer, and calling `signal(slotavail)`. If any processes happen to be waiting on the condition `slotavail`, one of them would be unblocked. If there is more than one process blocked, it is unspecified which one will be unblocked.

Notice the symmetry between `PutItem` and `GetItem`. Furthermore, the only explicit synchronization involves waiting for a slot to become available or an item to become available, and signaling when those conditions become true. There are no explicit synchronization statements to enforce mutual exclusion, unlike in our semaphore-based solutions.

Now, let’s look at the code for the Producer and Consumer processes. The

Producer enters a loop, repeatedly calling `Produce` to generate an integer, and then calling the monitor procedure `PutItem`. Notice how simple this code is. The Consumer is equally simple. It enters the loop, repeatedly calling the monitor procedure `GetItem`, and consumes that item that is returned by `GetItem`. Keep in mind that the Producer and the Consumer are processes, and in fact, this code allows for many producers and many consumers.

Let's see what happens if the Consumer were to run first. It tries to get an item by calling `GetItem`. Since `count` will equal 0 (given that it is initialized to 0), the Consumer will call `wait(itemavail)` and block. This is exactly the behavior we want, as there is nothing to remove from the buffer.

When the Producer runs, it will check whether `count` equals `N` (the size of the buffer), and since `count` equals 0 (the buffer is initially empty), it adds an integer to the buffer (which was produced by the call to `Produce` and passed to `PutItem`), increments `count`, and calls `signal(itemavail)`. Since the Consumer is blocked, waiting for this specific condition, the Consumer is unblocked.

The Consumer can then proceed and remove the item that was just added. It then calls `signal(slotavail)`, but as there are no processes that are waiting for this condition, the `signal` call does nothing. It is essentially “lost,” i.e., no record is kept that the signal happened. Notice that this is a key difference from the `wait` and `signal` operations of semaphores. In monitors, the condition variables do not have any memory; there is nothing to save and then later recall. A semaphore, on the other hand, has an

integer value associated with it, and each time `wait` and `signal` are called, the integer value is decremented and incremented, respectively.

Consequently, condition variables are really just *labels*. This makes the use of `wait` or `signal` in monitors very easy: `wait` simply causes a process to block, and nothing else; `signal` will cause a process that was blocked to unblock, or does nothing if there are no such processes, and nothing else.

There's one final point we must emphasize. Notice there is no *explicit* mutual exclusion. What happens if, while the Producer is in the middle of `PutItem`, that there is a context switch to, say, a second Producer that then also calls `PutItem`. The second Producer will actually block upon calling `PutItem` because of the monitor rule. This automatically provides mutual exclusion, which is thus *implicitly* part of the monitor. In fact, the entire monitor can be viewed as a critical section.

How is this mutual exclusion enforced? Since a monitor is a programming language construct, during compilation, the compiler adds instructions to enforce mutual exclusion. Consequently, the programmer does not have to be concerned with adding synchronization statements for mutual exclusion, which is very convenient. The compiler might accomplish this mutual exclusion by using semaphores, specifically a “*mutex*” semaphore, as we saw in the last chapter.

Let's elaborate on how synchronization works for a monitor. Figure 6.12 illustrates a room with a side waiting room as a metaphor for monitor synchronization. The room has a door with a lock, called the “*monitor lock*,”

whose default position is open. When a process calls a monitor procedure, it corresponds to trying to enter the room. If the door is unlocked, it is able to do so. Upon entering the room, the door closes behind it and the monitor lock automatically closes. While in the room, the process can execute the code of the monitor procedure. This corresponds to the process being “active” inside the monitor. Recall the monitor rule, that only one process can be active inside the monitor.

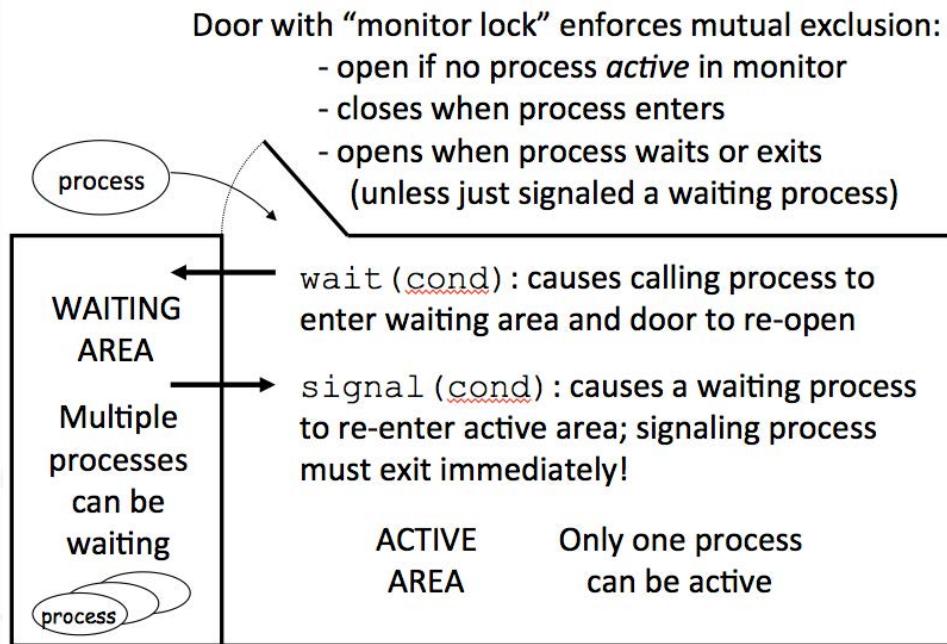


Figure 6.12: Room metaphor for monitor synchronization.

If there were a context switch to another process, and that process calls a monitor procedure, that process would block, given that the door is now closed. It could not get in the monitor, and only while in the monitor can

the process execute its code. The process would have to wait to enter the monitor, and when the monitor lock opens, that process can then enter.

When does the monitor lock open? It opens when the process that was executing inside the monitor is done and exits the monitor. At that point, the monitor lock opens, and the process that was waiting to enter is unblocked, and allowed to enter. Upon entering, just like before, the monitor lock will automatically close so that other processes trying to enter the monitor, i.e., processes that call any of the monitor procedures, are prevented from doing so. The process that just entered is now considered the one that is “active inside the monitor.”

Say that a context switch occurs to yet another process that calls one of the monitor’s procedures. That process would block, given that the monitor lock is closed, and the process would have to wait, just like the previous one. If the active process, the one already inside the monitor, were to call `wait` on a condition variable, that process would be moved to the waiting area, and would no longer be “active inside the monitor.”

At that point, the monitor lock automatically opens, allowing the process that was waiting outside to enter. The monitor lock automatically closes, and the process that just entered is now considered the one that is “active inside the monitor.” We do have a second process in the monitor, but as it is inside the waiting area, and because of this it is not considered active, the monitor rule is satisfied.

Now, say that the active process calls `signal` on the same condition

variable. This will automatically move the process that was waiting on that condition (and so waiting in the waiting area) into the monitor, making it “active.” We now have a problem: there are *two* processes active inside the monitor, which breaks the monitor rule. To resolve this situation, we can add an additional rule: When a process calls `signal`, it must then immediately exit the monitor. Consequently, the process that was moved from the waiting area to the active area is now the sole process in the monitor, and the rule that only one process be active in the monitor is satisfied.

If there were multiple processes in the waiting area, waiting for the same condition, and an active process called `signal` on that condition, only one process would be moved from the waiting area into the monitor. Which one is chosen is left unspecified. Furthermore, there can be multiple processes waiting on different conditions, and only one that corresponds to the condition being signaled will be allowed to enter.

Notice that it is the monitor lock that enforces mutual exclusion. Its automatic opening and closing relieves the programmer from having to add explicit mutual exclusion statements. Yet, the programmer can still add directives for “general synchronization” through the use of `wait` and `signal`. And, the semantics of `wait` and `signal` for monitors are simpler than those for semaphores, as there is no state (i.e., memory of past calls to `wait` and `signal`) involved. These features make monitors preferred for IPC.

We can now see that monitors, as a form of IPC, do indeed support both data transfer and synchronization, as required. Data transfer is supported

through the use of shared variables, which are accessed and modified via the monitor procedures. Synchronization is supported by both the monitor lock, for mutual exclusion, and by the `wait` and `signal` operations, for general synchronization.

However, there are issues with monitors that complicate their use. We've already encountered one, where if we have one process, P_1 , waiting on condition c , and P_2 signals condition c , then both P_1 and P_2 are running inside the monitor, which is a critical section, breaking mutual exclusion. This led us to adopt a second rule: when a process calls `signal`, it must then immediately exit. Technically, we have two processes running in the monitor, but only momentarily, and such that the exiting process cannot do any harm.

But how is this enforced? How can we prevent a programmer from not having any statements after a `signal` statement? We can rely on the fact that monitors are a programming language construct, and have the compiler check: If it finds any statements beyond a `signal` statement, it generates a compile time error. Notice that in our solution for the Producer/Consumer problem shown in Figure 6.11, the `signal` statements were indeed the last statements of the procedures before returning.

A second issue is that condition variables have no memory. Consequently, when `signal(cond)` is called, if no process is waiting on the condition `cond`, the signal is effectively lost. This is why, in our solution to the Producer/-Consumer problem using a monitor, we had to include the variable `count`, which indicated how many items were in the buffer. We did not have to do

this with our solution using semaphores because semaphores could be used for this counting purpose.

There are more advanced issues, such as: What happens if there are two monitors, and a process calls a procedure in one monitor, and while in that monitor, calls a procedure in the other monitor, and then calls `wait?` The monitor lock for the second monitor should clearly open, but what about that of the first monitor? What if another process then calls a procedure in the second monitor (which has its monitor lock open), which then calls a procedure in the *first* monitor. What should happen? These issues can be resolved in various *ad hoc* (and generally not very satisfying) ways, but they do add complexity and can lead to confusion.

Despite these issues, monitors bring added structure to IPC, making their use preferable to the use of the combination of semaphores and shared memory. Monitors localize critical sections and synchronization, making them easier to program and to reason about. However, keep in mind that monitor is a programming language construct, and so if you are using a programming language that does not support the monitor concept, you are out of luck.

6.6 Message Passing

Our third form of IPC is message passing. *Message passing* consists of two procedure calls, generally in the form of system calls, `send` and `receive`, as shown in Figure 6.13. The `send` system call causes a message to be deliv-

ered to a process identified as the *destination*, and the `receive` system call causes a message to be received from a particular process called the *source*. Importantly, `receive` will *block* if no message has arrived from the source (including if the source has yet to even send a message to the destination), i.e., there is nothing to receive yet, but the process will be made to wait for one.

<code>send(destination, &msg)</code>	Causes a message in the sending process's memory, at the memory address <code>&msg</code> , to be sent to a <i>destination</i> process
<code>receive(source, &msg)</code>	Causes a message sent by a <i>source</i> process to be received in the receiving process's memory, at the memory address <code>&msg</code>

Figure 6.13: Message passing operations: `send` and `receive`. Typically, there will also be a third parameter (not shown) that indicates the size of the message (which, for the sender, is the actual size of what should be sent, and which, for the receiver, is the acceptable size of what can be received).

We call this form of `receive` *synchronous* (i.e., a “synchronous receive”) because it will only return when a message is actually received (i.e., the two events, return from the `receive` call and reception of a message are synchronized, made to occur “at the same time”). On the other hand, the form of `send` we present is called *asynchronous* (i.e., an “asynchronous send”) since it does not wait, but rather returns immediately. (In general, an operation is considered synchronous if waiting is involved, otherwise it is asynchronous.)

These are the most common forms of `send` and `receive` (asynchronous, and synchronous, respectively) available in most systems. But, it does not have to be this way. We can have an “asynchronous `receive`” that does not cause blocking if a message has not arrived (and so the process would actually have to check whether the receive actually received a message). We can also have a “synchronous `send`,” which will block until the message sent is actually received by a process having called `receive`. Some operating systems offer all of these possibilities.

If message passing is truly an IPC abstraction, then it must support both data transfer and synchronization. Let’s see if it does. Data transfer is supported because the message, upon being sent, is copied from the process’s memory into a buffer (i.e., a temporary holding area) in the kernel until a process tries to receive the message, at which point the message is copied from the kernel buffer into the receiving process’s memory.

Synchronization is supported because, as described earlier, the form of `receive` is synchronous, i.e., it blocks to wait for a message. This satisfies the causal property that a process cannot receive a message until after it has been sent (and the logical property that it cannot receive a message that has not been sent). The programmer need not be concerned with implementing the synchronization, as it is provided by the `receive` call. Consequently, we see that message passing does indeed support both data transfer and synchronization, making it a valid IPC abstraction.

Let’s consider how the Producer/Consumer problem is solved using mes-

sage passing. This is shown in Figure 6.14. Both the Producer and the Consumer consist of `while` loops. The Producer calls `Produce` to generate an item, and places it into a variable called `item`. Note that this variable is local to the Producer (it maybe a global or local variable, but importantly, it is not a shared variable that would be available to the Consumer). The Producer then calls `send` to send the contents of the variable `item` to the Consumer. It then loops back, and repeats these two statements.

```
/* NO SHARED MEMORY */

Producer
int item;

while (TRUE) {
    item = Produce ();
    send (Consumer, &item);
}

Consumer
int item;

while (TRUE) {
    receive (Producer, &item);
    Consume (item);
}
```

Figure 6.14: Solution to the Producer/Consumer problem using message passing.

The Consumer calls `receive` to receive a message from the Producer, which gets placed in a variable called `item`, again noting that this variable is different from the one of the Producer; it is not a shared variable. If there is no message pending, the Consumer will block. Once a message arrives, the Consumer will be able to proceed and consume the item. It then loops back, and repeats these two statements.

There are number of striking features of this code. First, notice how *simple* it is compared to the prior solutions! This is because, most of the

complexity due to data transfer and synchronization is captured within the `send` and `receive` calls, leaving the programmer with just having to indicate what is to be transferred between which processes.

Secondly, note that there is no shared memory involved, which is another simplification. This is important, as it allows for implementation in environments where shared memory is not possible, such as in the network where the Producer runs on one machine and the Consumer runs on another. This is why message passing is so popular today, as most applications are built to run on the Internet. It also leads to safer code, as the ability to read and write share memory by separate processes can often lead to errors (such as race conditions).

One possible problem with our solution is the potential for the Producer to overwhelm the kernel with messages. Imagine the Producer running for a long time, sending many messages before the Consumer gets a chance to receive them. The kernel would have to provide a large amount of space to buffer all these pending messages. Given that there will be some limit as to how much the kernel can buffer, do we need to be concerned that there will either be an overflow, or that some messages may simply be discarded? Not necessarily.

Keep in mind that it is the kernel that controls the execution of processes, specifically, when to schedule a process (i.e., when it should be given the CPU, and for how long). Consequently, a solution to this problem is to have the kernel not allow a process to run if it has reached a limit in how many

messages have been sent but have yet to be received. In this way, the process is prevented from sending any more messages! This is essentially making the `send` synchronous when there is some maximum number (or maximum accumulated size) of outstanding messages. There are still subtle problems that need to be worked out (e.g., what happens if the Consumer never runs?), but the problem is manageable.

However, if the programmer is concerned with having too many outstanding messages, flow control between the Producer and the Consumer can be added, as shown in Figure 6.15. By *flow control*, we mean limiting the rate at which messages can be sent. This rate can be fixed to the rate at which the Consumer can receive messages.

Producer	Consumer
<code>int item;</code>	<code>int item;</code>
	<code>do N times {</code> <code> send (Producer, &item);</code> <code>}</code>
<code>while (TRUE) {</code>	<code>while (TRUE) {</code>
<code> receive (Consumer, &item);</code>	<code> receive (Producer, &item);</code>
<code> item = Produce ();</code>	<code> Consume (item);</code>
<code> send (Consumer, &item);</code>	<code> send (Producer, &item);</code>
<code>}</code>	<code>}</code>

Figure 6.15: Adding flow control.

This is achieved by having the Consumer first send N messages to the Producer. These messages will be “dummy” messages, in the sense that they contain nothing. However, they are sent as indicators that the Consumer

is ready to receive that many messages (one per dummy message). Consequently, at any point in time, there can be N outstanding messages between the Producer and the Consumer. Once the Producer has sent that many messages without any of them being received, it will block until the receiver has sent a dummy message to indicate that it can receive more. The number N of outstanding messages will determine how much parallelism is possible between the Producer and the Consumer. The larger N is, the longer either can run without having to wait for the other.

You might have noticed that the code in Figure 6.15 is not optimal in that the processes need not wait for messages, indicating permission to send, until they are actually ready to send. Figure 6.16 shows an optimized version of the code in Figure 6.15, where the Producer will first try to generate an item (which may take some time), and only when it is ready to send, does it check whether it has received a dummy message from the Consumer, which tells the Producer that the Consumer is ready to receive it. Likewise, as soon as the Consumer receives a message from the Producer, knowing it can consume it, it immediately sends a dummy message (indicating its willingness to accept another), without waiting for the message to actually be consumed (which may take time).

Just like with the previous IPC abstractions, there are issues with message passing that complicate matters. One is in the way that `receive` is expected to be used, such as in server processes. A server process is expected to get messages that contain requests from clients that are not known in advance.

Producer	Consumer
int item, empty;	int item, empty;
	do N times {
	send (Producer, &empty);
	}
while (TRUE) {	while (TRUE) {
item = Produce ();	receive (Producer, &item);
receive (Consumer, &empty);	send (Producer, &empty);
send (Consumer, &item);	Consume (item);
}	}

Figure 6.16: An optimization after having added flow control.

Consequently, it would help if it could express that it is willing to receive from any client, and once the message is received, it can then learn the name of the client process and direct a response appropriately. Consequently, we can imagine allowing for a “wildcard” as the source parameter in the `receive` call, signifying “receive from anyone,” and then perhaps learning the actual name of the source in some other way (as an extra parameter, or the return value of `receive`, etc.).

There is a more general problem with the naming of processes in both the `send` and `receive` calls. We indicated that a process sends a message by naming a destination process, and that a process receives a message by naming a source process. But what if the process does not know the name of the other process, as will often be the case? This is especially true if the processes are not related to each other (i.e., were not created by the same parent, or one did not create the other). Indeed, communication between

processes will typically not have been pre-arranged, and so a process would have to learn the name of the other process before it could communicate with it.

One solution is to avoid using process names in the `send` and `receive` calls, but rather to use “ports.” A *port* is essentially a place that a process can send messages to, which then get queued there, and then later, for another process to receive messages from that place. The post office, which is a message delivery system, uses mailboxes for this purpose, rather than delivering mail directly into the hands of the recipient. Just like a mailbox offers a more permanent place where a letter can be placed, independent as to whether the recipient is home or not, and even independent as to whether the name on the letter is correct or even present, a port offers a more permanent place to hold messages for processes whose names may or may not be known, and which can change.

A port is a “long-lived” object, and may correspond to a “*service*,” whereas a process may be short-lived, and dynamically connect to the service when necessary. For example, HTTP (the “Hyper-Text Transfer Protocol” used in web communication) is identified with the port “80,” which does not change. A web client process can send a message to that port (having also identified the name of a specific machine), without having to know the name of the web server process, and similarly, a web server process that actually provides the web service can receive messages from the same “port 80.” without having to know the name of the web client process that sent the

message. This level of indirection greatly facilitates communication between processes that are unrelated.

There is the issue of buffering messages in the kernel, and how to prevent a process from overwhelming the kernel with sent messages that are not received. We showed how this might be done (by having the kernel limit how many messages a process can send by preventing it from being scheduled for execution until some of the messages are removed by having been received). If the sending and receiving processes are on different machines, there are two kernels involved (one on each machine), and so there will need to be coordination between the kernels.

Despite these issues and others too many to mention – What happens if a message is lost (i.e., does not arrive within an expected amount of time)? What happens if a message that was considered lost actually arrives (i.e., it arrives beyond the expected time)? What happens if a process that is supposed to receive a message prematurely exits or is removed? – message passing is a very attractive abstraction for IPC. Because there is no shared memory, message passing is very natural for communication over networks. And because the programmer does not have to deal with shared memory, it is considered safer than the use of semaphores and shared memory, or monitors, which also relies on shared memory.

6.7 Summary

It is advantageous to structure programs as a set of cooperating processes for reasons of performance and modularity, but this requires the ability for processes to communicate with each other. IPC, or inter-process communication, is used for such communication. To support IPC, two mechanisms are required, data transfer and synchronization. Three important examples of IPC abstractions include semaphores and shared memory, monitors, and message passing. Each can be used to solve a classical problem, the Producer/Consumer problem

6.8 Exercises

1. What is meant by “processes that cooperate?”
2. Why structure a computation as a set of cooperating processes?
3. What is meant by “inherent parallelism” of a computation?*
4. How does parallelism promote performance?*
5. Can you give an example of how cooperating processes promote modularity?**
6. Given a single CPU, can a computation composed of cooperating processes run faster than the same computation structured as a single process (all single-threaded)?***

7. Given cooperating processes, what are meant by the following structures or relationships: pipeline? client/server? parent/child? Can you give actual examples of each?*
8. What is IPC?
9. What are the two mechanisms required by IPC, and what is meant by each of them?
10. Can you give an example of a mechanism that only achieves data transfer?* What about only synchronization?*
11. What is the Producer/Consumer problem?
12. In Figure 6.4, how is data transfer implemented?*
13. Why is synchronization needed?**
14. In Figure 6.5, what is the data transfer mechanism (or abstraction)?**
15. Is there synchronization: why or why not?**
16. Is synchronization actually needed, and if so, for what?**
17. In Figure 6.5, are there critical sections?***
18. Is it relevant how many producers there are, or how many consumers, and if so, why?**
19. In Figure 6.7, can you explain the role of all the semaphores?**

20. What is meant by “general synchronization vs. mutual exclusion”?**
21. In Figure 6.8, where is the race condition?**
22. Can you explain the role of all the semaphores?**
23. In Figure 6.9, what might happen if the `wait` statements in the Producer are interchanged?*** What about in the Consumer?***
24. What is a monitor?
25. What are the elements that comprise monitors?
26. How is data transfer achieved in monitors?**
27. How is synchronization achieved in monitors?**
28. How is mutual exclusion implemented in monitors?***
29. What are condition variables in monitors, and how are they used?**
30. What does the `wait(cond)` operation do?*
31. What does the `signal(cond)` operation do?*
32. What is meant by the property that “only one process can be active inside a monitor”?* How is this property implemented?***
33. Can you explain how the code in Figure 6.11 works?**
34. In Figure 6.12, how is data transfer achieved?**

35. How mutual exclusion achieved?**
36. How general synchronization achieved?**
37. Are there any system calls?***
38. In Figure 6.13, what is meant by the “door” and “monitor lock,” and how do they work?**
39. What is meant by the “Active Area”?* How about the “Waiting Area”?*
40. How does a process enter the “Active Area”?* How about exit the “Active Area”?*
41. What causes a process to enter the “Waiting Area”?* How about exit the “Waiting Area”?**
42. What is the issue/problem with monitors that arises when a process P_1 is waiting on condition c and another process signals c ?**
43. What is a solution to the problem in the previous question?***
44. What is meant by “Condition variables have no memory”?**
45. If `cond` has no memory, then what is being supplied to `wait(cond)` and `signal(cond)`?***
46. If a process calls `signal` on a condition but there is no process waiting for that condition, what is meant by “the signal is lost”?**

47. How are monitors different from semaphores?***
48. How are monitors different from semaphores + shared memory?***
49. Can monitors be implemented with just semaphores?***
50. Can semaphores be implemented with monitors?***
51. In what way are monitors more *powerful* than semaphores?***
52. What is message passing?
53. What is the data transfer mechanism?
54. What is the synchronization mechanism?*
55. Can you explain how `send(destination, &msg)` works?**
56. Can you explain how `receive(source, &msg)` works?**
57. What is the role of the kernel in the above?*
58. Can you explain how the code in Figure 6.15 works?**
59. Are there system calls in Figure 6.15?**
60. Is it necessary that the message passing calls, send and receive, be atomic?***
61. Can you explain how the code in Figure 6.16 works?***
62. What is the optimization in Figure 6.16?***

63. What is the issue with message passing regarding who (or what) should messages be addressed to?***
64. What are ports (or mailboxes)?**
65. Why is it convenient to allow processes to receive messages from any process?***
66. What is the issue with kernel buffering of messages?***
67. Why is message passing a good paradigm for IPC over networks?**
68. Why is message passing safer than shared memory paradigms?***
69. Is message passing more powerful than monitors?***
70. Is message passing more powerful than semaphores?***
71. Can semaphores be implemented with monitors, and if so, how?***
72. Can semaphores be implemented with message passing, and if so, how?***
73. Solve the Dining Philosophers Problem (as described in Exercise 55 of Chapter 5) using monitors.***
74. Solve the Dining Philosophers Problem, using message passing.***
75. Solve the Reader/Writers Problem (as described in Exercise 62 of Chapter 5) using monitors.***
76. Solve the Reader/Writers Problem using message passing.***



Chapter 7

Deadlock

We've seen the value of organizing a computation as a set of cooperating processes. However, there is one especially difficult problem that can sometimes arise. This is the problem of deadlock.

What is deadlock? A *deadlock* occurs when there is a set of processes that are permanently blocked. The unblocking of one relies on the progress of another. But since none can make progress, none will be unblocked, and the deadlock condition will remain.

7.1 Resource Allocation Graphs

Consider the *resource allocation graph* in Figure 7.1. We have two processes, A and B, indicated by circles, and two resources, X and Y, indicated by squares. If a process is *holding* a resource, meaning that the resource was

allocated to that process, we draw an arrow from the resource to the process. If a process is *waiting* for a resource, meaning that a request was made by the process to have the resource allocated to it but because the resource is being used by another process and thus busy it cannot be allocated at the moment and the process must wait, we draw an arrow from the process to the resource.

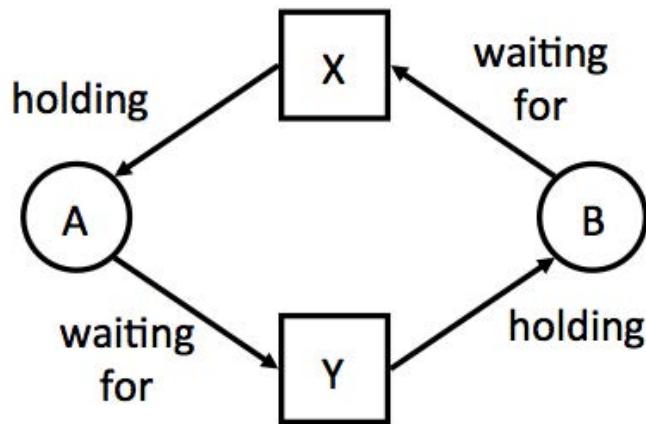


Figure 7.1: A simple resource allocation graph, also known as a “wait-for” graph.

In the graph in Figure 7.1, we see that process A is holding resource X and waiting for resource Y. Process B is holding resource Y and waiting for resource X. Since each process is waiting for the other to give up its held resource, but this will only happen when each process is able to get the other resource, they will wait forever. This is an example of a deadlock.

Deadlocks happen in real life. Figure 7.2 shows an intersection with a set of cars, each of which has progressed to the middle of the intersection. It is

important to be able to identify what are the processes, and what are the resources.

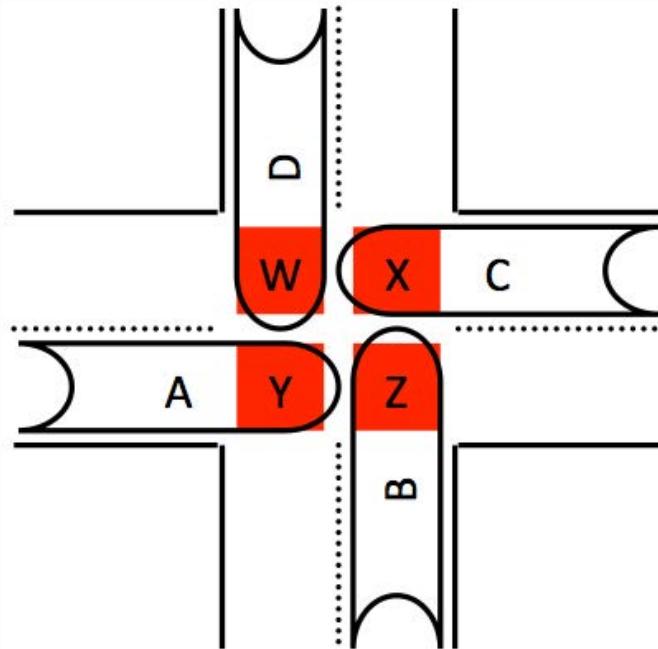


Figure 7.2: Cars deadlocked in an intersection.

The processes are the cars. The resources are the parts of the intersection, specifically the four squares that comprise the intersection. In order for a process to cross the intersection, it needs the two consecutive squares in the direction in which it is going. In the example, car A has square Y but cannot proceed until it gets square Z. Car B has square Z but it cannot proceed until it gets square X. Car C has square X but cannot proceed until it gets square W. And car D has square W but cannot proceed until it gets square Y.

We can convert this situation into a resource allocation graph, which is

shown in Figure 7.3. We can now see a pattern. A deadlock exists when we can find a *cycle* in the resource allocation graph. We see a cycle in Figure 7.3, and in Figure 7.1.

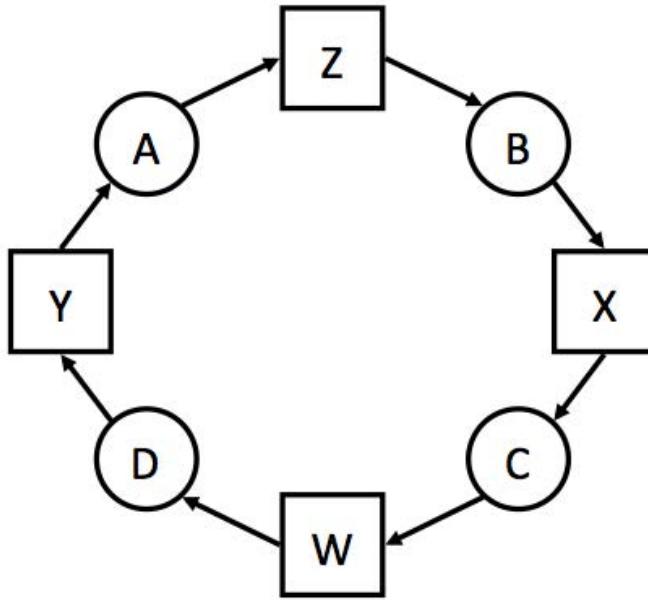


Figure 7.3: The resource allocation graph for the cars in the intersection.

Figure 7.4 shows an example of memory, where resources are given *incrementally*, and how allocations can lead to deadlock. Considering memory of 200 MB, all of which is initially available. Say process P_1 requests 80 MB. Since 80 MB is available, P_1 is allocated that memory. Say process P_2 then requests 70 MB. Again, since 70 MB is available, P_2 is allocated that memory. Next P_1 requests 60 MB. However, of the 200 MB, only 50 MB are free. Consequently, P_1 must wait. P_2 is still able to run and requests 80 MB. Again, since only 50 MB are free, P_2 must wait. Since the only way

that either process will be unblocked is when memory becomes available, but since the processes themselves are holding that memory, neither will give their memory up because they are blocked, and we have a deadlock.

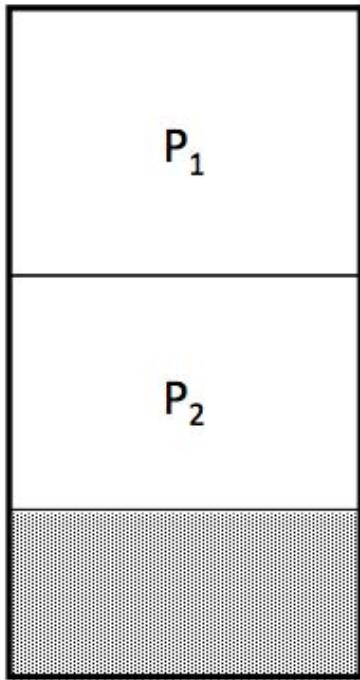


Figure 7.4: A memory that is allocated incrementally.

7.2 The Four Conditions for Deadlock

There is a well-developed theory regarding deadlock. We know that for deadlocked to be possible, four conditions must be present. They are shown in Figure 7.5.

These conditions are all *necessary*. Remove any one of them, and dead-

1. Mutual exclusion: only one process may use a resource out of time
2. Hold-and-wait: a process must be holding a resource while waiting for another
3. No preemption: once a process has a resource, it cannot be forcibly taken away
4. Circular wait: the waiting processes must form a cycle, each process waiting for one of the others

Figure 7.5: The four conditions for deadlock.

lock will not be possible. This observation leads to a strategy: to prevent deadlock, just make sure one of the conditions is removed.

There are three broad strategies for attacking the deadlock problem. The first is called *deadlock prevention*, whereby deadlock is made impossible by simply removing one of the necessary conditions. The second is *deadlock avoidance*, where we try to avoid getting into situations that lead to deadlock. And finally, the third is *deadlock detection*, where, rather than trying to prevent or avoid deadlocks, we allow them to happen, with the idea that we can then detect them, and then try to somehow resolve the situation at that point. In this chapter, we will go through each of the strategies.

7.3 Deadlock Prevention

In deadlock prevention, the idea is quite simple. Since all of the four conditions for deadlock are necessary, just to remove any one of them, and a

deadlock simply cannot happen, i.e., deadlock is prevented. Let's go through each of the conditions, and see how viable it is to not allow that condition to be present.

The first condition is mutual exclusion. We can remove this condition by allowing resources to be shared, rather than their being forced to be used one at a time. However, for some resources, it only makes sense to use them one out of time. For example, a memory that is private to a process cannot be shared with other processes. Or, a critical section of code simply cannot be shared; it must be used by one process at a time. A printer cannot be shared in that, if a document is being printed by one process, another process can't be using it to print another document *at the same time*. Consequently, we may not have the ability to simply remove the mutual exclusion condition.

How about the second condition, hold-and-wait? This condition says that a process can be holding a resource while it is waiting for another. We can remove this condition by forcing processes to request all the resources they may need simultaneously, and waiting until all of them are free. In this way, a process is not holding some resources while waiting for others.

This may sound reasonable, but in general, processes do not know precisely what are all of the resources that they may need. A process will generally learn this as time goes along. Secondly, a process will not be using all of the resources that it may need at all points in time. And so, this can be a wasteful approach. However, if indeed a process knew of all the resources that it needed, and we are not concerned that they may be used inefficiently,

this approach would certainly work. In fact, this approach is used in “mission critical” situations, such as in controlling a nuclear power plant, where resources are pre-allocated such that when they are needed, they are ready to be used. It may be expensive, but it works in that it is guaranteed that there will be no deadlocks.

The third condition is no preemption. We can remove this condition by allowing preemption, i.e., simply allowing resources that have been allocated to be taken away. However, this is often inconvenient and undesirable. Imagine a printer being used to print a long document, and the printer is then taken away to start printing another document before it finishes printing the first. The user that submitted the first print request will not be very happy with this situation! Also, allowing preemption may simply not be possible. For example, if there is no clock interrupt, we cannot implement preemption of the CPU; the process holding it must voluntarily give it up. So, getting rid of this condition is typically either inconvenient or impossible.

Finally, we come to the fourth condition: circular wait. The way to get rid of this condition is to prevent cycles. One clever way of doing this is to order all of the resources, which can be done by assigning each of them a unique integer ID. Then, have processes follow a rule such that a new resource can only be accessed if its ID is greater than the IDs of all processes that it is currently holding. If not, the process has to give up some or all of the resources it is holding so that the rule can be followed. In other words, have all processes order their acquisitions of resources according to increasing ID

number.

This will actually work, but there are issues. For example, it is often not possible to assign every resource an ID, at least at the beginning of time, because some resources only come into existence on demand, e.g., a block of memory created by a memory allocator. Even if this is not an issue, acquiring resources in a particular order, especially when the process does not know beforehand what all the resources will be, can be inconvenient and inefficient. But, if these are not issues for a particular system (as may be the case for mission critical systems), this approach can certainly be used, and deadlocks will be avoided.

Consider our cars-in-an-intersection problem, as shown in Figure 7.6. If our theory of deadlock is truly general, then it should apply to this problem. If we remove one of the conditions for deadlock, we should be able to prevent a traffic jam. A common way of solving the traffic problem in real life is to add a traffic light. In doing so, the traffic jam in Figure 7.2 cannot happen. Adding a traffic light must then correspond to removing one of the conditions for deadlock. Which one?

It's not mutual exclusion, because we can't change the laws of physics and allow two cars to share the same physical space. How about hold-and-wait? Indeed, this is the condition that is removed, as a traffic light will allow a car to acquire the two squares it needs to cross the intersection, rather than getting stuck in the middle of the intersection by holding one square and waiting for the other. For completeness, let's look at the other

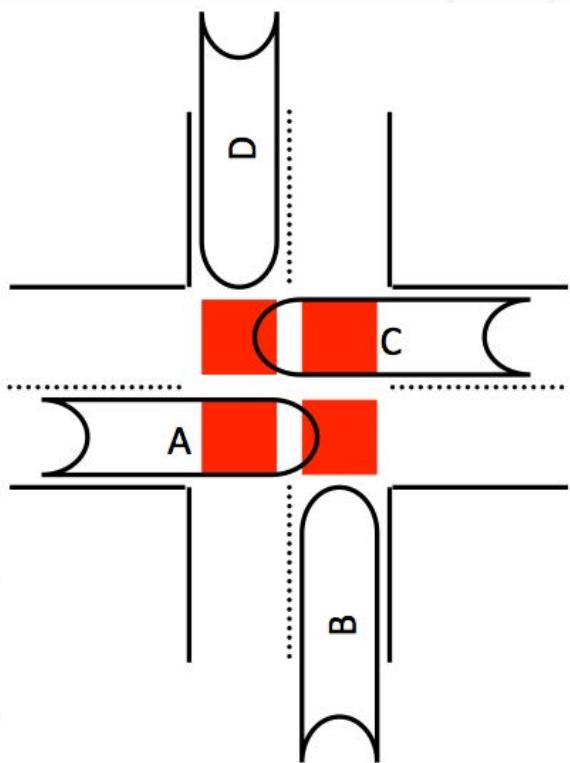


Figure 7.6: Cars in an intersection governed by a traffic light.

two conditions. We did not get rid of no preemption, as once a car enters the critical section and occupies one square we cannot take the square back (unless we forced the car to back up). Finally, the circular-wait condition is irrelevant, because we've removed hold-and-wait.

This last point brings up a curious relationship between the two conditions, hold-and-wait and circular wait. Circular wait *depends* on hold-and-wait, but not vice versa. If we removed hold-and-wait, we've removed circular wait. But if we remove circular wait, we can still have hold-and-wait. Consequently, it is good to consider these two conditions separately, as there may be situations where it is convenient to remove circular wait, but difficult or impossible to remove hold-and-wait. But since all we need is to remove one of them, we are guaranteed that deadlocks will be prevented if we remove just circular wait.

7.4 Deadlock Avoidance

In deadlock avoidance, the idea is to avoid situations that *may* lead to deadlock. It is a “softer” approach compared to deadlock prevention. The idea is to apply *selective prevention*, by removing a condition only when deadlock is possible.

This approach works especially well with incremental resource requests. By this, we mean that resources are asked for in increments, and we do not satisfy the request (at that moment in time) if there is the possibility it will

lead to a deadlock. This approach requires the constraint that the maximum resource requirements for each process must be known; they would have to be declared by the process, or we might assume some global maximum for all processes. Note that this does not mean the process will ask for all the resources, but only that the process will never request more than the given maximum.

The *Banker's algorithm*, by Dijkstra, follows this approach. It assumes there are a fixed number of processes and a fixed number of resources, where each process has zero or more of the resources allocated to it. A specific allocation of resources to processes determines the system state, of which there can be two possibilities, either safe or unsafe.

A *safe* state is one where deadlock is absolutely avoidable. Specifically, there exists a certain order of execution of the processes such that deadlock can be avoided. The goal then is to find this order of execution. An *unsafe* state is one where deadlock is possible, i.e., that it may not be possible to avoid deadlock. Note that this does not mean that a deadlock will definitely happen, but just that there is no guarantee that it will not happen.

Figure 7.7 shows a diagram that illustrates the meaning of safe, unsafe, and deadlock states. If the system begins in the area labeled “safe,” which corresponds to the system being in the safe state, it is guaranteed to remain safe as long as we do not venture outside that area. Consequently, at each step, we ask whether the state will change from safe to unsafe. If so, we delay that move. The move would correspond to allocating a resource to satisfy a

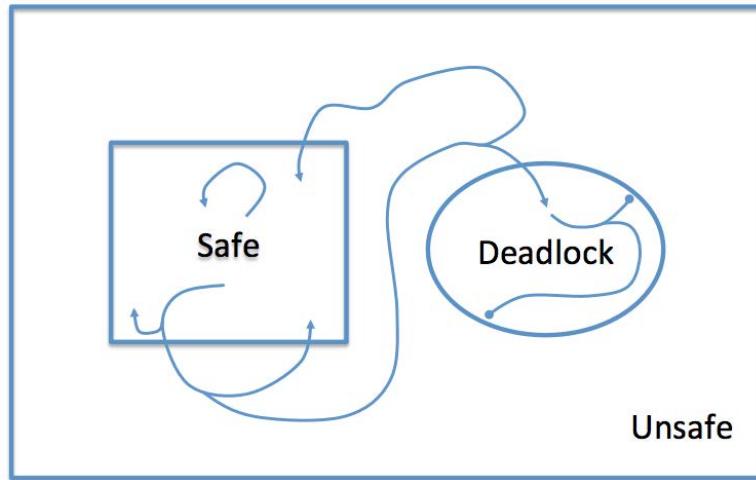


Figure 7.7: An illustration of safe, unsafe, and deadlock areas.

process's request. Delaying that move corresponds to making a process wait until another process gives up some resources.

Notice that if we venture outside the safe area, we are in the unsafe area, i.e., the system state is unsafe. This means that a deadlock is possible, but *not necessarily guaranteed*. But once in the unsafe area, it is possible to end up in the deadlock area. The key observation is that, if we never venture into the unsafe area, we can never end up in the deadlock area. Consequently, the idea is to remain in the safe area.

This is not unlike telling a child that they can play on your side of the street, and not go across the street. If they do, it does not mean that something bad will definitely happen, but only that it may happen. In other words, crossing the street means entering an unsafe area. As long as the child stays in the safe area, nothing bad will happen. (If only life were that

simple!)

Let's look at the Banker's algorithm in detail. There are three data structures that are maintained:

1. a process/resource claim matrix (a 2D array)
2. a process/resource allocation matrix (a 2D array)
3. a resource availability vector (a 1D array)

For a given assignment of values to the elements of these three data structures, the system can be characterized as either being safe or unsafe.

The characterization is determined as follows. By default, we assume that the system begins in a safe state (if not, we must obtain more resources before we can proceed). We then ask the question: Is there an *ordering* of processes, i.e., a schedule in which they will run, such that first process is able to run to completion, return its resources, which can then be used by a second process and in doing so will allow that process to run to completion, return its resources, and so on for all processes? *If an ordering can be found such that all the processes are able to complete, we call that a safe state; otherwise it is an unsafe state.*

Let's look at a few examples. Figure 7.8 shows a claim matrix, an allocation matrix, and an availability vector, populated with values. The particular assignment of values determines whether the state is safe or unsafe. But, to make this determination, we must ask whether there is a ordering or sequence of processes such that if they were made to run in that order, each would be

	Claim				Allocation				Avail-ability	Total
	P ₁	P ₂	P ₃	P ₄	P ₁	P ₂	P ₃	P ₄		
R ₁	3	6	3	4	1	6	2	0	0	9
R ₂	2	1	1	2	0	1	1	0	1	3
R ₃	2	3	4	2	0	2	1	2	1	6

Figure 7.8: A process/resource claim matrix, a process/resource allocation matrix, and a resource availability vector, with values that determine this to be a safe state.

able to complete, without the possibility of having to be blocked because a potential resource request cannot be satisfied.

In making this determination, we have to consider all possible orderings of processes until we find one. Let's begin with choosing process P_1 to execute first. P_1 has declared that it may claim up to 3 units of resource R_1 , 2 units of resource R_2 , and 2 units of resource R_3 (notice that resources come in units, or increments). P_1 is currently allocated 1 unit of R_1 , and 0 units of R_2 or R_3 . Consequently, during its execution, it may ask for *up to* 2 units of R_1 . This is not to say it will definitely do so (it might only ask for one unit, or even none), but only that it may. And if it does, we see that it will block, because the availability vector tells us that there are no available units of R_1 . Consequently, choosing an ordering where P_1 runs first would lead to an unsafe state, and so we discard all such orderings.

How about choosing P_2 to run first. P_2 has declared that it may claim up to 6 units of resource R_1 , 1 unit of resource R_2 , and 3 units of resource

R_3 . It is currently allocated 6 units of R_1 , and 1 unit of R_2 , and 2 unit of R_3 . Consequently, during its execution, at most, it may ask for 1 more unit of R_3 , and nothing else. Since 1 unit of R_3 is available, P_2 could run to completion. Consequently, an ordering that begins with P_2 looks promising. But we won't know until we see that all the other processes are also able to complete.

Once P_2 completes, it will return all of its resources: 6 units of R_1 , 1 unit of R_2 , and 2 units of R_3 , where the availability vector has the values 6, 2, and 3 for R_1 , R_2 , and R_3 , respectively. Given the resources just made available by P_2 , we could select P_1 to run next, as it would be able to complete its execution even if it requested the maximum of what it claims it may need. Consequently, we have an ordering with the first two processes determined, P_2 and P_1 .

After P_1 completes, it will return all of its resources: 1 unit of R_1 , and 0 of R_2 and R_3 (if it had requested additional ones during its execution, those would be returned too), and so the availability vector will have the values 7, 2, and 3 for R_1 , R_2 , and R_3 , respectively. It is not difficult to see that, if we select P_3 to run next, it will be able to complete, and then selecting P_4 to run last, all will be able to complete without the possibility of blocking.

This entire analysis that we did to find at least one ordering that would allow all processes to complete, without the potential for blocking, is what is needed to be able to characterize the particular configuration of values in Figure 7.8 as a safe state. If we were not able to find any such ordering,

then we would say that that specific configuration of values corresponds to an unsafe state.

		Claim				Allocation				Avail-ability	Total	
		P ₁	P ₂	P ₃	P ₄	P ₁	P ₂	P ₃	P ₄			
R ₁	P ₁	3	6	3	4	P ₁	2	5	2	0	0	9
	R ₂	2	1	1	2	P ₁	0	1	1	0	1	3
	R ₃	2	3	4	2	P ₁	1	1	1	2	1	6

Figure 7.9: A process/resource claim matrix, a process/resource allocation matrix, and a resource availability vector, with values that determine this to be an unsafe state.

Consider Figure 7.9. This is an example of an unsafe state. The reason is that no process can definitely run to completion. For example, if we choose P_1 to run first, it may block asking for 1 unit of R_1 . The same is true if we were to choose P_2 , or P_3 , or P_4 . But, there are no units of R_1 available. Note that we don't know if any of these processes will exercise their right to make these requests; in fact, if they don't, they may all be able to run to completion. But we don't know that! Consequently, since deadlock is possible, even though not necessarily certain, we deem this an unsafe state.

We can use the Banker's algorithm to avoid unsafe states. Given that we assume the system begins in a safe state, we are guaranteed that there is some ordering that would allow processes to complete. If this is not the case, we obviously can't make this assumption, and would have to obtain more resources so this assumption holds. Regardless, we *must* be able to assume

the system starts in the safe state.

Consequently, we allow the first process in that ordering to run. We know that whichever resources it requests, we will be able to satisfy all those requests. During the execution of that first process, *we can now allow a context switch* to occur. In selecting the next process to run, we would ask, might allowing this process to run, at this time, cause the system to go to an unsafe state? If so, we would not allow that process to run next. And so we would consider another process, and ask the same question. At worst, we know that we can always allow the first process to continue running, as allowing that process to run completion will not result in an unsafe state.

A common misconception with this approach is that the system changes states over time, alternating between safe and unsafe. It does not; in fact, the whole idea is that it must not! The system begins in the safe state, and for each scheduling decision, the scheduler determines whether the particular choice under consideration will lead to an unsafe state. If so, that choice is discarded. Consequently, the system remains in a safe state throughout. This is guaranteed because, by definition, a safe state is one where all the processes are able to complete under a particular ordering. We can just follow that ordering, or consider different orderings as time goes on, as long as those other orderings do not lead to unsafe states.

7.5 Deadlock Detection and Recovery

This approach to dealing with deadlocks is to do nothing special to prevent or avoid them. Rather, we accept that they may happen, and if they do, we will then deal with the situation. Periodically, the system will try to detect that a deadlock occurred. If so, it will then try to do something about it. Or, it may even do nothing!

The reasoning behind this approach is that deadlocks rarely happen. Consequently, to equip a system with so much machinery that will add to its size and complexity, and yet be used so infrequently, is not worth the trouble. If a deadlock happens, then we'll just deal with it in a special and perhaps *ad hoc* way.

To detect the deadlock, the system constructs a resource allocation (“wait-for”) graph, as was discussed at the beginning of this chapter. We saw examples of resource allocation graphs in Figure 7.1 and Figure 7.3 To determine that a deadlock occurred, it will check for cycles in the graph. If a cycle is found, a deadlock exists, and it involves those processes and resources that comprise the nodes in the cycle. This requires identifying all resources, tracking their use, and periodically running the detection algorithm.

Once a deadlock is detected, the system can attempt to recover from it. One approach to recovery is to terminate all the deadlocked processes. This will certainly remove the deadlock, but it is drastic and can be costly. A less severe way of doing this is to terminate deadlocked processes one at a time,

and to see if the deadlock comes back or not. The order in which processes are terminated may be made dependent on how costly it is to terminate each particular process. The less important processes would be terminated first.

We must also consider the resources that are part of the deadlock. If processes are terminated, the resources that were being used might be left in an inconsistent state. For example, a process may have partially written a file (the resource in this case), leaving it in a state that is unusable by other processes. Another example would be an interrupted message or file transfer. Somehow, there would have to be a way of restarting the transfer, or doing away with it completely (and ignoring or undoing what was already transferred). Suffice it to say that this is a difficult problem, and resolving each situation in an *ad hoc* way adds complexity to the system such that it may outweigh what was gained by not doing deadlock prevention or avoidance.

The most extreme solution is to do absolutely nothing. In this case, the system ultimately relies on the *user* to determine that something is wrong, and act accordingly. In fact, in this “approach,” the user is given both the responsibility for detecting deadlocks and for recovering from them. For example, observing that a process (or a group of processes) is “frozen,” (e.g., a mouse click in a window has no effect), the user would surmise that a deadlock has occurred. This is the “detection” part of the approach. At that point, the user would take action. The most common would be to manually kill the process, or in the worst case, reboot the entire system. This is the “recovery” part of the approach! As extreme as this may sound, this is the

approach that is most likely expected for the operating system running on your laptop!

7.6 Summary

A deadlock occurs when a set of processes are mutually waiting for each other; the progress of any one of them depends on that of another, but since all are blocked, no progress can or will ever be made. For a deadlock to be possible, four conditions must be present: mutual exclusion, hold-and-wait, no preemption, and circular wait. Each is necessary: remove any one condition, and deadlock becomes impossible.

There are three broad strategies for dealing with deadlocks: deadlock prevention, deadlock avoidance, and deadlock detection and recovery. In deadlock prevention, one of the necessary conditions is explicitly removed. In deadlock avoidance, a condition is removed, but only at certain times and temporarily. In deadlock detection and recovery, deadlocks are simply allowed to occur, with the expectation that they will then be detected and the system will attempt to recover. In its extreme form, the responsibility for detection and recovery is placed on the user.

7.7 Exercises

1. What is meant by “deadlock”?

2. When discussing deadlocks, what is meant by a “resource”?*
3. In Figure 7.2 what are the processes and what are the resources?*
4. In Figure 7.4, what are the processes and what are the resources?*
5. What are the four conditions of deadlock?
6. What is the “mutual exclusion” condition?
7. What is the “hold-and-wait” condition?
8. What is the “no preemption” condition?
9. What is the “circular wait” condition?
10. Are the conditions independent from each other; why or why not?**
11. What is meant by deadlock prevention?
12. What is meant by deadlock avoidance?
13. What is meant by deadlock detection and recovery?
14. How is deadlock prevention different from deadlock avoidance, and can you give an example?**
15. Which is the easiest to implement; which is the hardest?**
16. In deadlock prevention, for each of the four conditions for deadlock, how can they be prevented, and can you give an example of each in a real situation?**

17. In Figure 7.6, which condition is being prevented, and why?*
18. What kind of operating system would use this technique?***
19. In deadlock avoidance, why is it considered “selective prevention”?*
20. What is meant by incremental resource requests?
21. What is meant by “need maximum resource requirements”; can you give an example?*
22. What kind of operating system would use this technique?***
23. In the Banker’s Algorithm, what is meant by a “safe state”?* What about “unsafe state”?*
24. In Figure 7.7, what is being conveyed by the illustration?*
25. What is a process/resource claim matrix?*
26. What is a process/resource allocation matrix?*
27. What is a resource availability vector?*
28. What is the process ordering test used by the Banker’s algorithm?*
29. In Figure 7.8, why is the state safe?*
30. In Figure 7.9, why is the state unsafe?*
31. In Figure 7.10, the intersection is governed by the rule: at most 3 cars allowed. Which condition is being avoided?**

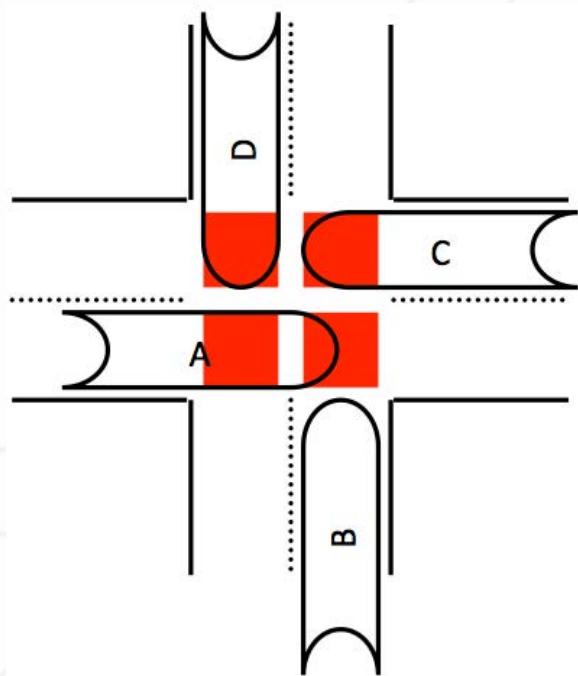


Figure 7.10: An intersection governed by the rule: at most 3 cars allowed.

32. What is the key justification for allowing deadlocks to happen in Deadlock Detection and Recovery?
33. Why do most general-purpose operating systems use this technique?**
34. What would be an example of an operating system that cannot use this technique?***
35. For deadlock detection, what is the purpose of the wait-for graph?*
36. How can an operating system recover from deadlock?*
37. Why is terminating all deadlocked processes costly?**
38. What is the reasoning behind terminating processes one at a time, and what are some of the complications?**
39. In your solution for the Dining Philosopher's Problem in Exercise 55 of Chapter 5, modify it so the deadlocks are avoided.***
40. In your solution for the Readers/Writers Problem in Exercise 62 of Chapter 5, modify it so the deadlocks are avoided.***



Chapter 8

Memory Management

Memory management refers to how to allocate and free portions of the memory. Allocation of memory occurs when a new process is created, or when a process requests more memory. Freeing of memory occurs when a process exits, or when process no longer needs memory that it requested.

8.1 Process Memory Areas

Each process requires memory in order to store its code, data, stack (of activation records), and perhaps other items for other uses. Recall that a process memory has at least three such areas:

- *text*, which stores the code;
- *data*, which stores static variables and heap; and

- *stack*, which stores activation records that contain, amongst other items, automatic variables for called procedures.

There may also be shared memory areas, i.e., memory shared with other processes.

These memory areas have distinguishable characteristics. They each have a size, which may be fixed or variable, i.e., they may be able to grow or shrink. They have permissions, such as read, write, and execute, indicating what are the permitted operations that can be applied to their elements, e.g., words and bytes.

These memory areas, and their elements in particular, must be addressable. An *address space* is a set of addresses to access in memory. Typically, an address space is linear and sequential, with addresses ranging from 0 to $N - 1$ for a memory of size N .

8.2 Process Memory Layout

Figure 8.1 shows the address space and layout of a process's memory, containing the text, data, and stack areas, and showing the addresses that mark their boundaries. A text area of size X will be located at addresses 0 to $X - 1$. The data area will immediately follow, from addresses X to $X + Y - 1$, where Y is the size of the data area. The stack area, of size Z , has addresses $N - Z$ to $N - 1$.

Notice that the stack area is at the far opposite end of the process's

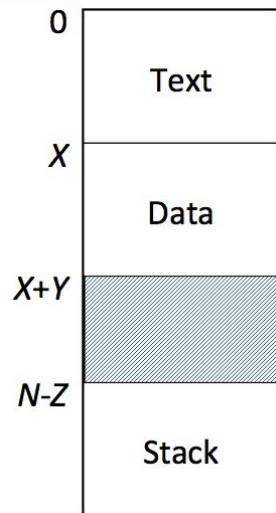


Figure 8.1: Address space and layout of text, data, and stack areas within a process's memory.

memory. It grows (is allocated) in the opposite direction than the data area, towards smaller addresses. This is done so that the data and stack areas can be as distant from each other as possible, leaving free addresses, and thus memory, between them. This allows both data and stack areas to grow without colliding with each other until there is absolutely no more memory, making the most efficient use of the memory.

8.3 Role of the Compiler

This view of memory is especially relevant for the compiler. The compiler must have a model of the process's memory so that it can generate memory addresses when it compiles a program. After all, it is the compiler that de-

cides where instructions and data variables should go, and for those instructions that reference variables, it must be able to generate memory addresses for them to include in the machine code.

The compiler must also be aware that the data and stack areas can change in size. That data area will change with dynamic memory allocation instructions as expressed in the program, and which only occur when the program runs. The stack area will change whenever there is a call to a procedure, in which case an activation record is generated that is allocated on the stack, and whenever there is a return from a procedure, in which case the activation record is freed from the stack. The compiler is charged with generating instructions to support this growth and shrinkage.

However, what the compiler does not generally know is the size of the physical memory in which the program will run. One reason to know this would be to know where to place the stack, i.e., at the high end of the address space. The compiler also does not generally know what areas of the physical memory have been already allocated when the program is to eventually run. In fact, every time the program runs, the portions of physical memory that are being used will generally be different. There is no way for the compiler to be able to predict this when it compiles a program, despite that it must generate memory addresses.

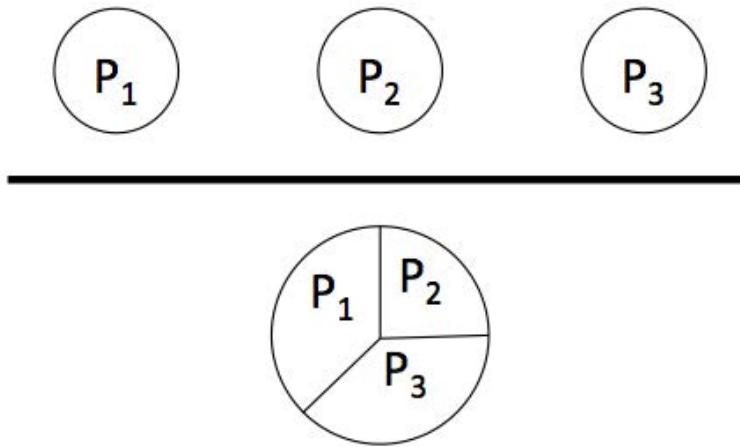


Figure 8.2: Each process seemingly has its own CPU, but underlying this view is the reality that the CPU is shared, with its time apportioned to each process.

8.4 Memory Space and CPU Time

Keep in mind that one of the goals of the operating system is to support *multiple* processes. To support programs to run “simultaneously,” the operating system implements the process abstraction, and the kernel causes the single CPU to be shared amongst all runnable processes. This creates the illusion that that each process has its own CPU, albeit operating at a fractional speed. This is the view we presented in the previous chapters, as is depicted in Figure 8.2. But, in addition to CPU time, a process requires memory space.

Consequently, the operating system provides an abstraction for processes whereby each process has, not just its own CPU, but also its own memory.

But this is just an illusion. In reality, the kernel is causing the CPU as well as the memory to be shared amongst all processes. While there is a single physical memory, each process would get a portion of that memory, making it seem as if each process had its own memory, albeit of fractional size.

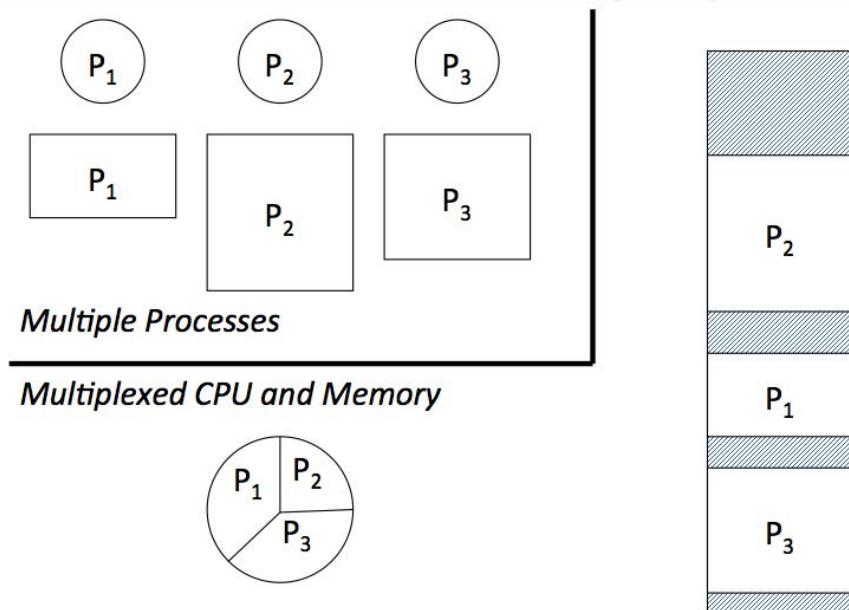


Figure 8.3: Expanded view of the abstraction and realization of processes, where each process seemingly has its own CPU and memory, but underlying this view is the reality that the CPU is shared and memory are shared, with CPU time and memory space apportioned to each process.

Within the allocated memory for a process, it would then be subdivided into text, data, and stack areas, with a “free” area between data and stack, as shown in Figure 8.3. Note that this will not be our final solution to how memory is organized, as you might already see problems, especially inefficiencies, with the approach as just described. In good time, these problems

will be addressed. For now, let us assume the model presented so far.

Let's consider the problems of sharing the physical memory in more detail. We're faced with the constraint that, if a process is given the CPU, in order to actually use it, the process must also be in memory. Consequently, these two resources, CPU and memory, must be made available for use at the same time.

8.5 Swapping

One approach to sharing the memory is to make use of an external storage device, such as a disk, and shuffle processes between the physical memory and the disk such that whenever a process is to use the CPU, it is brought into memory, while all the other processes are kept on the disk. In this way, the process gets to use the entire physical memory, and whenever there is a context switch to another process, the current process in memory is “swapped out,” and the process chosen to run is “swapped in.”

How practical is this approach? Let's assume that context-switching time is 10 microseconds. Let's also assume that a process can be transferred from the disk at a rate of 10 MB per second. Consequently, to load a one megabyte process would require 100 milliseconds, which is 10,000 times the time to do a CPU context switch! This is clearly too much overhead, and would certainly break the illusion of simultaneity.

So, we're forced to consider a more complex solution whereby multiple

processes are simultaneously kept in memory, and not just one. Context switching would only occur between the processes that are in memory. If all processes do not fit in memory, some will be stored on the disk, and periodically, there will be transfers between processes in memory and those on the disk so that all processes eventually have an opportunity to be in memory, and thus run. For the time being, we will ignore the second aspect of managing memory that involves swapping processes to and from the disk. We will just focus on managing the physical memory, and assume that all the existing processes fit in the physical memory.

Our discussion will proceed in three parts. First, we will discuss the issue of *memory management* (memory allocation algorithms), which determines how physical memory is allocated and freed for use by processes. Second, we will consider the concept of a *logical memory*, which is needed to develop the compiler's model of memory, and which will support *segmentation*, a logical view of memory that benefits the programmer. Finally, we will discuss *virtual memory*, which deals with the fact that our physical memory is indeed limited, and that we will have to rely on external storage devices to support large memories for processes. This will support the idea of paging, which moves fixed size units of memory between the physical memory and external storage device. Throughout we will discuss mechanisms and policies to support these ideas.

8.6 Memory Allocation Algorithms

We begin with an example of memory management. Consider a physical memory that is completely empty, as shown in Figure 8.4. We view this memory as having a single very large “hole” to be “filled in.” Over time, areas of the memory get allocated, as shown in Figure 8.5.

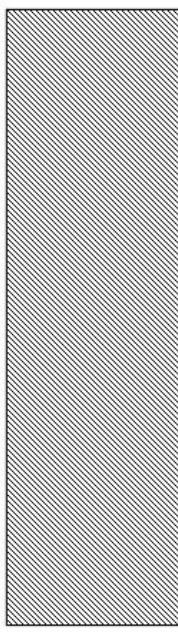


Figure 8.4: An empty memory, considered a single very large hole.

To allocate memory of a certain size, we have to find a hole large enough to contain that allocation, i.e., a hole of size that is at least that of the requested allocation, as shown in Figure 8.6. In general, the hole will be larger than the required allocation size (unless there were a perfect fit, which is very rare, so we will always assume imperfect fits unless stated otherwise).

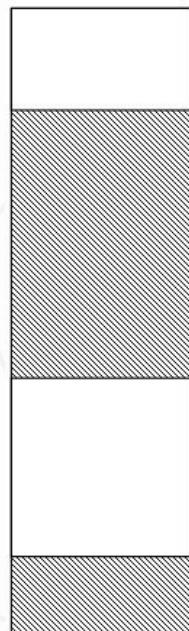


Figure 8.5: A memory where, after allocations and de-allocations, is left with two allocated areas and two holes.

Consequently, a portion of the original hole is filled, and this will typically leave a smaller *residual* hole, as shown in Figure 8.7.

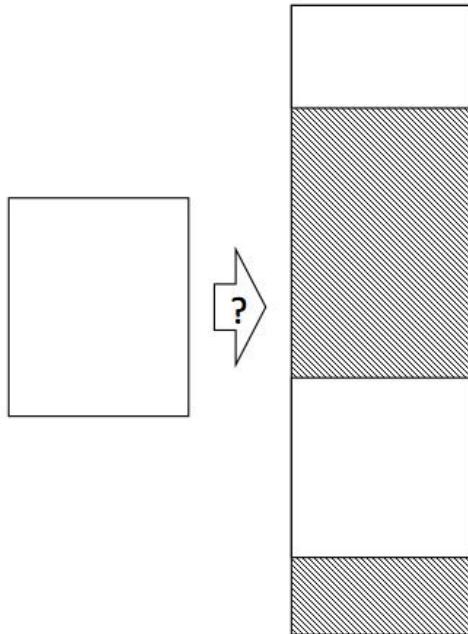


Figure 8.6: Attempting to fit a request for memory in a hole.

When the allocated memory is no longer needed, it is released or freed. The freed area of memory will create a new hole, as shown in Figure 8.8. If it is next to an existing hole (or two), the holes may be coalesced to form one large hole, as shown in Figure 8.9.

In general, when memory is allocated, there will be multiple holes to choose from (as any of them can accommodate the allocation), as shown in Figure 8.10. The question then becomes, which one should be selected?

There are number of possible algorithms.

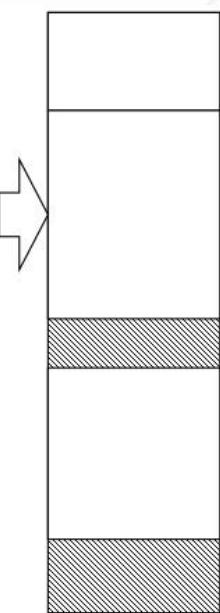


Figure 8.7: Finding a hole, i.e., making an allocation, leaving a smaller residual hole.

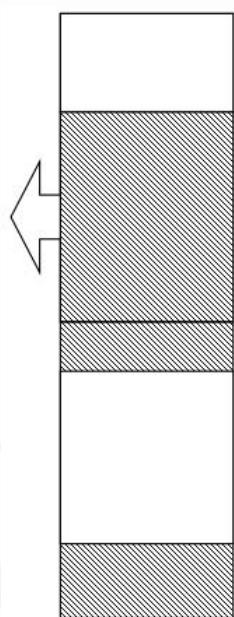


Figure 8.8: Freeing a block of previously allocated memory, resulting in a new hole, next to an existing hole.

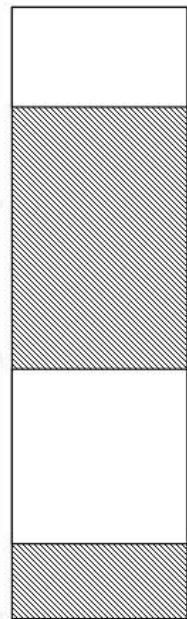


Figure 8.9: The result of coalescing holes that are next to each other to form one larger hole.

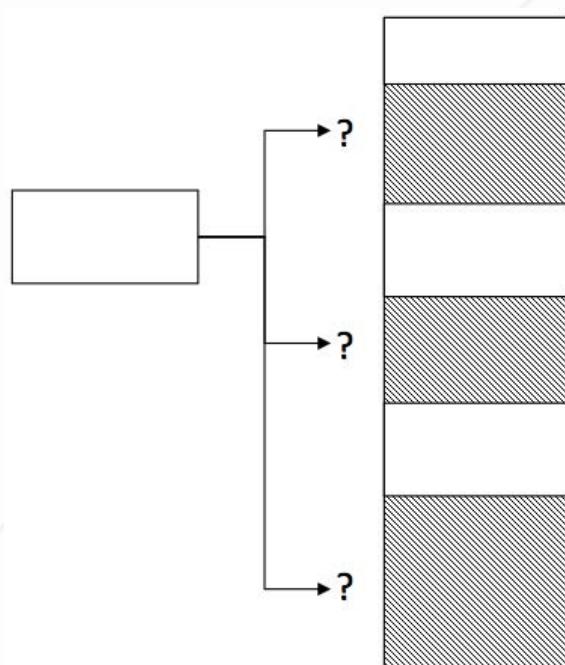


Figure 8.10: Given a memory request, which hole should it go in?

8.7 First Fit Allocation

The *First Fit* algorithm will select the first hole that is found, as shown in Figure 8.11. If the search is from the top of the memory, then as soon as a hole is encountered that is large enough for the allocation, that hole is used. A portion is allocated, and a smaller residual hole will be left over. First Fit is both simple and fast.

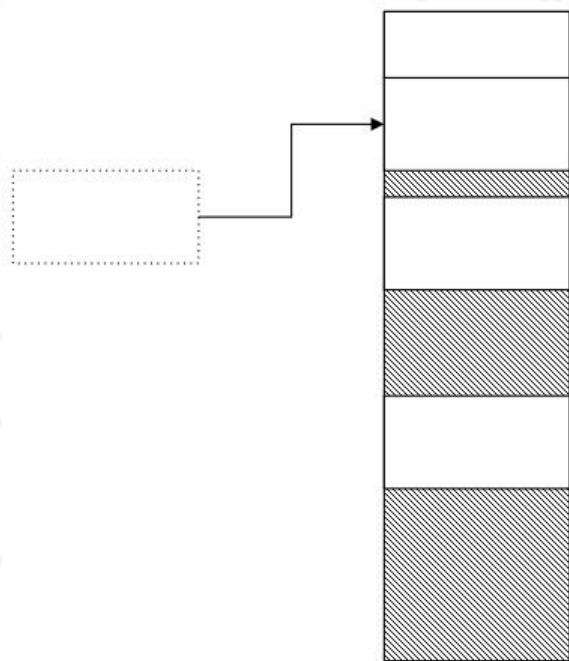


Figure 8.11: First Fit selects the first hole that will fit the memory request.

One might imagine that this will cause small holes to eventually develop near the top of the memory. Consequently, we can add a pointer that always points just beyond the last allocated area of the memory, and that the next

search will begin from that point rather than the top of the memory. This would spread the allocations and the resulting smaller holes that are left over more evenly over the entire memory. This variation of First Fit is called *Next Fit*.

Perhaps we can improve upon our selection of holes. After all, is the first hole that is found always the best? And what do we mean by “best?” One definition of best is the *smallest* hole that will satisfy the allocation. This may be viewed as good because it leaves larger holes available to satisfy larger requests that would otherwise not be able to be allocated.

8.8 Best Fit Allocation

This is precisely the way the *Best Fit* algorithm works. It checks every hole and determines which is the smallest that will still satisfy the request, as shown in Figure 8.12. That hole is then used for the allocation. However, the residual hole that it leaves will tend to be very small.

Best Fit sounds good, but does it perform well (never mind whether it is optimal)? Comparing it with First Fit, Best Fit has the disadvantage that every hole must be checked before an allocation is made. And for a very large memory, e.g., one or more gigabytes, it would not be unusual for there to be tens, or even hundreds, of thousands of holes. Consequently, checking every hole may take a significant amount of time.

Secondly, given that the residual holes resulting from Best Fit tend to

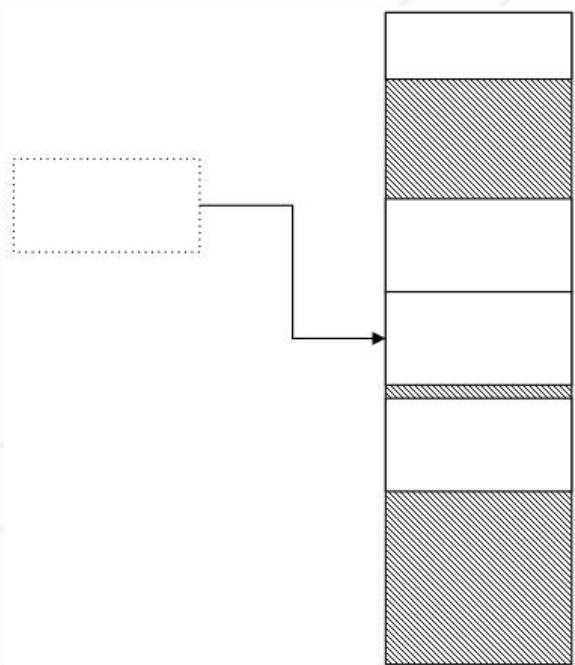


Figure 8.12: Best Fit finds the smallest hole that will satisfy the request.

be very small, they may be so small that they are unusable, leading to a highly fragmented memory with a significant portion of it that is unused and unusable.

8.9 Worst Fit Allocation

If the highly fragmented memory that results from Best Fit does not sound good, we can try the opposite approach. Rather than finding the “best” hole, find the “worst” one. The Worst Fit algorithm works as follows. It checks every hole, and determines which is the *largest* that will still satisfy the request, as shown in Figure 8.13. That hole is then used for the allocation. The residual holes that it leaves will tend to be larger than the ones left by Best Fit.

How good is Worst Fit? Like Best Fit, it suffers in that every hole must be checked before an allocation is made, which can take a long time. It remains debatable whether the resulting fragmented memory of residual holes is any better than that produced by Best Fit. However, Worst Fit has a further disadvantage that large holes are broken up first, and thus later requests for very large allocations will get denied, whereas they would not be denied with Best Fit.

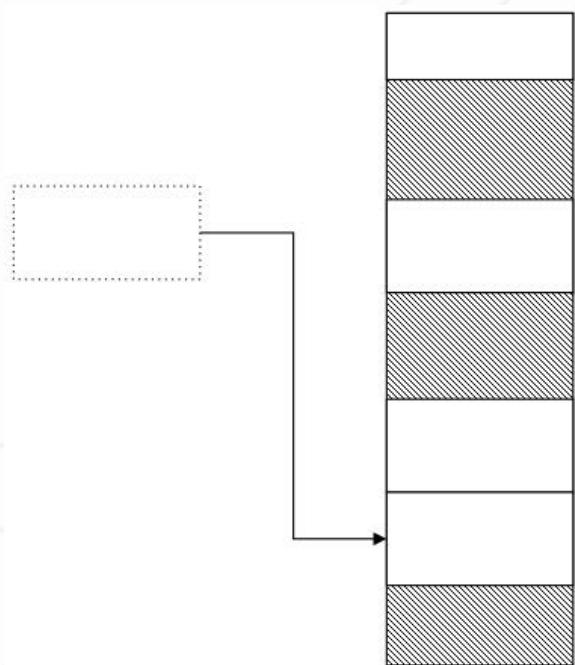


Figure 8.13: Worst Fit finds the largest hole that will satisfy the request.

8.10 Which Allocation Method is Best?

Answering this question of choosing a memory allocation algorithm, in a robust way, is not easy. So much depends on the memory request patterns (allocations and de-allocations), which depend on the workloads, and it is difficult to generalize these behaviors. Furthermore, memory behavior is typically more complicated than we have considered so far. For example, when a memory allocation is made, is it possible that it might be required to grow or shrink later on? If so, knowing this in advance would help as then a larger hole might be favored.

One way of resolving this question is to consider the tradeoff of memory space overhead vs. allocation time overhead. If we somehow better understood the quality of the various fit methods, we could determine the resulting overhead in memory space, i.e., the average amount of memory wasted by each algorithm. Similarly, we can consider the average time spent each algorithm consumes in finding a hole. We can then apply a technology trend argument: it is better to waste space than it is to waste time. After all, we can always buy more memory (up to a limit, of course), but we can just replace our CPU with a faster one. Or, we might argue that buying more memory is cheaper than buying a faster CPU. Given such an argument, we favor the faster algorithm, rather than the one producing more efficient memory usage.

Given that we may favor speed over space efficiency, and given that our

knowledge of how well the various algorithms do in terms of space efficiency is weak, we would choose First Fit. In fact, the most common algorithm that is used is indeed First Fit (or Next Fit). It is faster (in practice) than Best Fit or Worst Fit, and has no bias in the sizes of the residual holes that result.

8.11 Fragmentation

Eventually, memory will become *fragmented*. By fragmented, we mean that the memory will have many holes, some of which may even be unusable because they are too small. Unfortunately, fragmentation is unavoidable, and it is generally considered wasteful, though not always.

We distinguish between two types of fragmentation: external and internal. *External fragmentation* is unallocated space, and corresponds to the holes that we have been discussing. *Internal fragmentation* is unused space that is within an allocated block (in contrast to external fragments, which are outside allocated blocks). Figure 8.14 shows a memory with both internal and external fragments.

Why would fragments be *inside* of allocated blocks? Consider an allocation of memory that is larger than what was requested, and so part of it will be unused, i.e., an internal fragment. The reason for the larger allocation might be because that size of allocation is easier, or more convenient, or the only possible one due to a hardware constraint, than the one requested. Another reason might be an intentional one, because there is an expectation

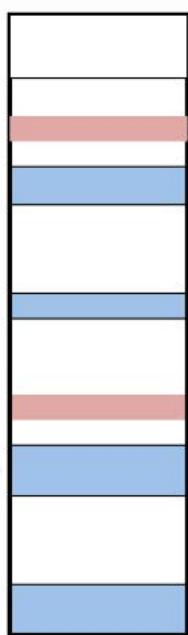


Figure 8.14: A memory with both external (in blue) and internal (in red) fragments.

that the original area will not be adequate, e.g., will need to grow. Consequently, internal fragmentation can come in handy for growth. As another example that we've already seen, consider a large single block of memory that is allocated for process, with the expectation that it will contain areas for text, data, and stack. Between the data and stack will be unused memory, which would correspond to what amounts to a very large internal fragment. This allows either the data area or stack area to grow, i.e., make use of the internal fragment's space.

Note that when we have internal fragmentation, that memory cannot be allocated to the other processes, which makes it fundamentally different than external fragmentation. The memory within external fragments can still be allocated (unless they are too small, making them unusable).

8.12 Compaction vs. Breaking Memory Requests into Smaller Pieces

Consider the following problem: say that there is a request to allocate a block of memory, but that there is no hole large enough to satisfy that request, as shown in Figure 8.15. However, there may be a significant amount of unused space considering the cumulative space of all external fragments. (Note that the space of the internal fragments cannot be allocated, as, they are already allocated.) Might it be possible to use those fragments in some way?

One approach is compaction: to compact the memory by moving the

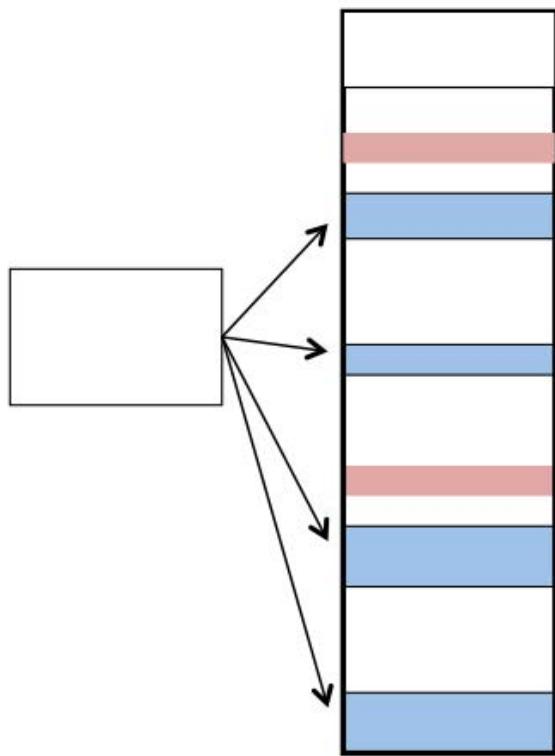


Figure 8.15: A memory request that cannot be satisfied by any of the existing holes, i.e., external fragments.

space of all the external fragments so they are next to each other, and coalescing them into one large fragment, as shown in Figure 8.16. This hole may now be large enough to accommodate the memory request.

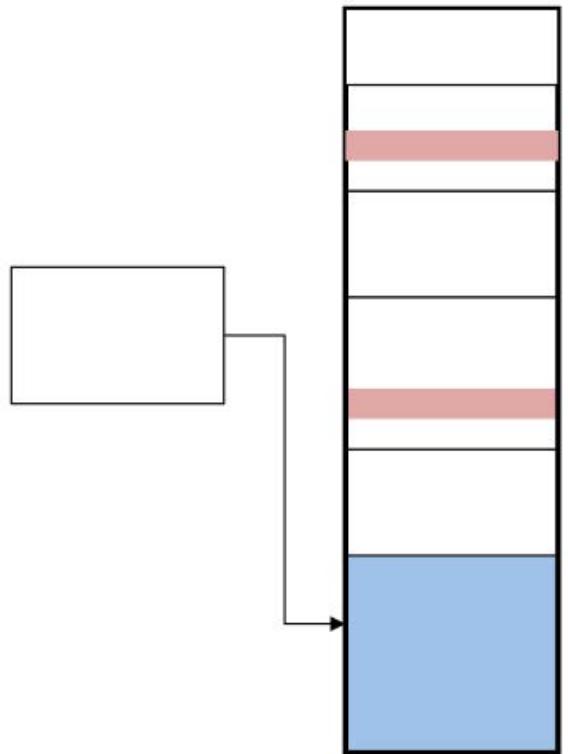


Figure 8.16: Compacting the memory by moving all the external fragments next to each other and coalescing them.

Compaction is a simple idea, and may even seem like a good idea, but in reality, it is not very effective because it is very time consuming. Considered that in “moving” all of the external fragments so they are next to each other requires copy already allocated areas to other parts of the memory. In fact, since external fragments are generally distributed throughout the memory, a

compaction will require a copying of the majority of the memory. Memory operations are relatively expensive. If it takes 100 nsec for a main memory reference, then the amount of time to read and write a large portion of a 1 GB memory (assuming memory accesses in 4-byte words) can take many seconds!

Another approach to our problem of there not being a large enough hole for our memory request is to break up the request into smaller pieces. Then, perhaps the smaller pieces will be able to fit in the existing holes, as shown in Figure 8.17. On the other hand, we are now introducing more complexity, by somehow having to break up requests into smaller units.

Which is the better approach? When we consider the time vs. complexity tradeoff, given the immense amount of time required for compaction, we may be willing to admit extra complexity.

But is breaking up memory requests a realistic solution? Can we reasonably ask the programmer to change their memory requests into smaller pieces, and if so, how small should they be? Or, might this be something that can be done automatically, on behalf of the programmer? We will see that, indeed, this can be automated, and despite the complexity it adds to the memory system, both in hardware and in system software, it is worth it. But, we will have to wait until the next few chapters to see how this is done.

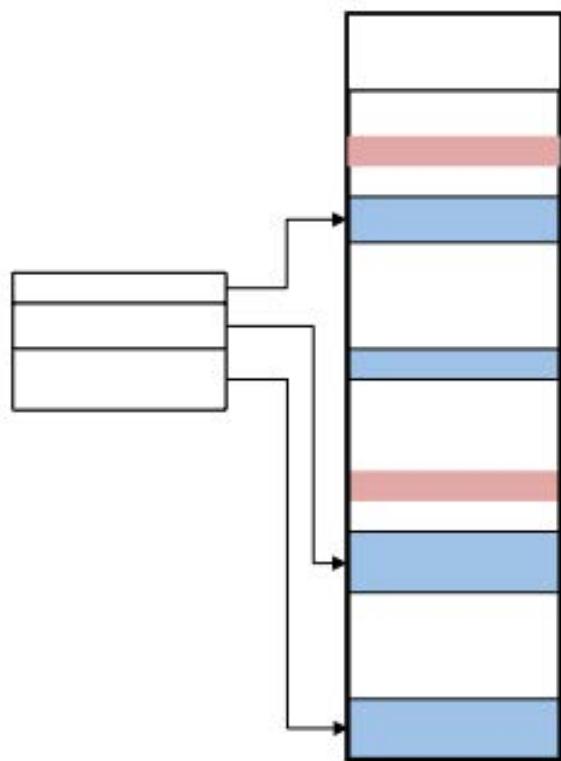


Figure 8.17: Breaking up a memory request into smaller pieces.

8.13 Memory Usage Analysis

How much of our memory will actually be fragmented? There are a few analytical methods that will give us an idea.

Let's first ask the question: Given n allocated blocks of memory, how many holes will there be? Consider the allocation of memory as shown in Figure 8.18. We see six blocks of allocated memory, which we will simply call blocks, and between some of them, three empty areas that are holes. Some blocks will be right next to each other, as when a new block is allocated, it will generally be next to an already allocated block. But we will never have two holes next to each other, because if we did, we would simply coalesce them.

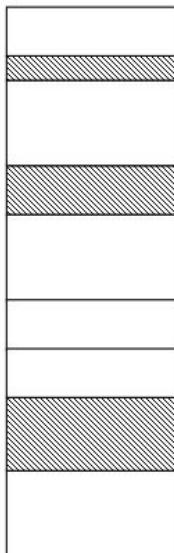


Figure 8.18: A memory with 6 blocks of allocated memory and 3 holes.

8.14 The 50% Rule

Can we express the number of holes as a function of the number of blocks, on average, $m = f(n)$, where m is the number of holes, and n is the number of blocks? The answer is yes, due to a result by Donald Knuth. It is called *the 50% rule*, where

$$m = n/2 \quad (8.1)$$

To derive this result, consider that there are three types of blocks:

Type A: block has hole on each side. Let a = number of A blocks.

Type B: hole on just one side. Let b = number of B blocks.

Type C: no holes on either side. Let c = number of C blocks.

Figure 8.19 shows examples of blocks of types A, B, and C.

Since the number of blocks is n , we have that

$$n = a + b + c \quad (8.2)$$

Since each A block is adjacent to two holes, and each B block is adjacent to one hole, and each C block has zero adjacent holes, the number of holes can be expressed as

$$m = (2a + b)/2 \quad (8.3)$$

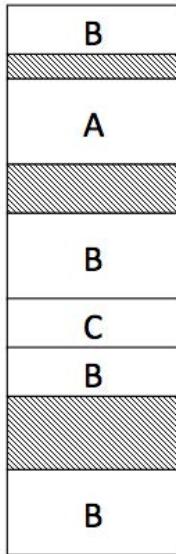


Figure 8.19: Blocks of allocated memory, of types A, B, and C.

We divide by two because each hole is adjacent to two blocks, and so, to not double count, we must divide by two.

For example, if we consider Figure 8.19, we see that the number of blocks $n = 6$, the number of A blocks is $a = 1$, the number of B blocks is $b = 4$, and the number of C blocks is $c = 1$. Our formula for the number of holes tells us that there are three holes ($m = ((2 \times 1) + 4)/2 = 6/2 = 3$), which is precisely the number of holes in Figure 8.19.

Consider what happens during de-allocation. When a block is freed, the number of holes changes as follows based on the type of block it is:

Type A: decreases by 1

Type B: stays the same

Type C: increases by 1

We can now express the following probabilities, given a block is freed:

$$\text{Prob}(\#\text{holes decrease} \mid \text{block is freed}) = a/n \quad (8.4)$$

$$\text{Prob}(\#\text{holes increase} \mid \text{block is freed}) = c/n \quad (8.5)$$

When a block is allocated, the number of holes will decrease by one if there is a hole that is an exact fit. Otherwise, the number of holes will stay the same. Let $p = \text{Prob}(\text{the hole is not an exact fit})$. We can then express the following probability, given an allocation:

$$\text{Prob}(\#\text{holes decrease} \mid \text{block is allocated}) = 1-p \quad (8.6)$$

$$\text{Prob}(\#\text{holes increase} \mid \text{block is allocated}) = 0 \quad (8.7)$$

We now make the following assumptions regarding *equilibrium*: the system is in equilibrium once a point is reached where the number of holes and number of allocated blocks stays constant. This assumption implies the following probabilities:

$$\text{Prob}(\text{block is allocated}) = \text{Prob}(\text{block is freed}) \quad (8.8)$$

$$\text{Prob}(\#\text{holes increase}) = \text{Prob}(\#\text{holes decrease}) \quad (8.9)$$

The $\text{Prob}(\#\text{holes increase})$ can be expressed as follows:

$$\begin{aligned} & \text{Prob}(\#\text{holes increase} \mid \text{block is freed}) \times \text{Prob}(\text{block is freed}) \\ & + \text{Prob}(\#\text{holes increase} \mid \text{block is allocated}) \times \text{Prob}(\text{block is allocated}) \\ & = (c/n) \text{Prob}(\text{block is freed}) + 0 \text{Prob}(\text{block is allocated}) \end{aligned} \quad (8.10)$$

The $\text{Prob}(\#\text{holes decrease})$ can be expressed as follows:

$$\begin{aligned} & \text{Prob}(\#\text{holes decrease} \mid \text{block is freed}) \times \text{Prob}(\text{block is freed}) \\ & + \text{Prob}(\#\text{holes decrease} \mid \text{block is allocated}) \times \text{Prob}(\text{block is allocated}) \\ & = (a/n) \text{Prob}(\text{block is freed}) + (1 - p) \text{Prob}(\text{block is allocated}) \end{aligned} \quad (8.11)$$

Setting these equal and using $\text{Prob}(\text{block is allocated}) = P(\text{block is freed})$
(by our equilibrium assumption):

$$c/n = a/n + 1 - p \quad (8.12)$$

therefore,

$$c = a + n(1-p) \quad (8.13)$$

Recall that $n = a + b + c$, therefore

$$c = n - a - b \quad (8.14)$$

Setting these equal, we have

$$a + n(1-p) = n - a - b \quad (8.15)$$

therefore,

$$np = 2a + b \quad (8.16)$$

Recall that $m = (2a + b)/2$, therefore

$$2m = 2a + b \quad (8.17)$$

Setting these equal

$$np = 2m \quad (8.18)$$

Therefore

$$m = (n/2)p \quad (8.19)$$

If all our fits are imperfect, then $p = 1$, and we arrive at the 50% rule:

$$m = n/2 \quad (8.20)$$

If all of our fits are perfect, then $p = 0$, and there will be no holes:

$$m = (n/2)0 = 0 \quad (8.21)$$

8.15 The Unused Memory Rule

Now that we have the 50% rule, we can derive another (perhaps even more) useful result, which tells us the total amount of memory due to holes.

Let

b = the average size of blocks

h = the average size of holes

$k = h/b$ = the ratio of average hole-to-block size

Given a memory of size M , it must be that

$$M = mh + nb \quad (8.22)$$

where m is the number of holes, and n is the number of blocks. We can find the fraction of memory due to holes as

$$f = mh/M = mh/(mh + nb) \quad (8.23)$$

Assume that all allocations are imperfect fits. We can use the 50% rule, which tells us that $m = n/2$, to express the fraction of memory due to holes as

$$f = mh/(mh + 2mb) = h/(h + 2b) = k/(k + 2) \quad (8.24)$$

Therefore, the fraction of memory due to holes can be expressed very concisely as $f = k/(k + 2)$, where k is the ratio of average hole-to-block size. This is the *Unused Memory Rule*.

Let's consider some sample values for k . If $k = 1$, this means that the average hole size equals the average block size, and in this case, the amount of memory due to holes is $1/(1 + 2) = 33\%$. Consequently, just due to the fact that blocks will not perfectly fit in holes, they'll be enough statistical variation such that even if the average hole size is the same as the average block size, 33% of our memory will be contained in holes. That is a large fraction of memory!

If $k = 2$, this means that the average hole size equals twice the average block size, and in this case, the amount of memory due to holes is $2/(2 + 2) = 50\%$. And if $k = 3$, this means that the average hole size equals three times the average block size, and in this case, the amount of memory due to holes is $3/(3 + 2) = 60\%$.

In general, f increases with increasing k : the larger the average hole size is to the average block size, the larger is the fraction of memory due to holes,

and as $k \rightarrow \infty, f \rightarrow 1$. Alternatively, f decreases with decreasing k : the smaller the average hole size is to the average block size, the smaller the fraction of wasted memory, and so, as $k \rightarrow 0, f \rightarrow 0$.

8.16 The Buddy System

These derivations tell us that variability is the source of our problems: variable-size allocations cause fragmentation. Given this, why not have pre-sized holes to reduce the variation?

In the extreme, we can consider same-sized holes. Since all holes are the same, allocation is very easy; just pick any hole. This may be too inflexible, as the holes may be too small, and thus unless blocks are small, they will not fit into the holes. Note that we will still have internal fragmentation, but at least the allocation of memory will be simplified.

A less extreme approach is to have a variety of sizes, such as small, medium, and large. With more flexibility comes more complexity. Which of the sizes be, and how many of each?

This brings us to an approach suggested by Donald Knuth, called “The Buddy System.” The idea is to partition memory into power-of-two size “chunks,” which refer to holes if they are unallocated and blocks if they are allocated. Given a request to allocate a memory block of size r , we carry out the algorithm as shown in Figure 8.20.

First, the memory begins as one large hole, or chunk. To allocate a

```

allocate(r) // find chunk larger than r
while(r <= sizeof(chunk)/2)
    divide chunk into 2 equal-sized buddies
allocate the chunk

free(chunk) // free the chunk
while(buddy is also free)
    coalesce

```

Figure 8.20: Allocation and freeing in the Buddy System.

memory block of size r , we must first find a chunk of at least size r . If no such chunk exists, then the allocation fails. But, if we do find a chunk, we first divide it in into two smaller chunks, called *buddies*. We then check if one of the buddies will be satisfactory for the request. If not, then we allocate the entire chunk for the memory request, and we're done. But if a buddy will suffice, then we repeat, using the buddy as the candidate chunk. We keep doing this until we have found the smallest power-of-two sized chunk that will satisfy the request.

To free an allocated chunk, we simply mark it free. If its buddy also happens to be free, we coalesce them, and apply the same freeing operation to the a new chunk, recursively. If the buddy is not free, we stop.

Let's illustrate the algorithm with an example. Figure 8.21 shows a memory of size 8 MB that is completely free; it is considered one large chunk.

Say there is a request to allocate 900 KB. We first have to find a free chunk that will satisfy that request; certainly, the existing 8 MB chunk will

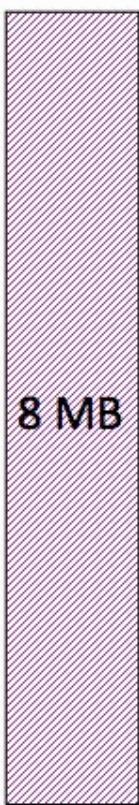


Figure 8.21: One large free chunk of size 8 MB.

do so. However, before allocating it, we break it into two buddies, or two 4 MB chunks, and ask whether one of those chunks will do. The answer is yes, but rather than allocating the 4 MB chunk, we break that chunk into two buddies, or two 2 MB chunks, and ask whether one of those chunks will do. The answer is yes, but rather than allocating the 2 MB chunk, we break that chunk into two buddies, or two 1 MB chunks, and ask whether one of those chunks will do. The answer is yes, but rather than allocating the 1 MB chunk, we break that chunk into two buddies, or two 1/2 MB chunks, ask whether one of those chunks will do. The answer is *no* (finally!), and so we go back to the 1 MB chunks and allocate one of them. Call the allocated chunk A. Our memory now looks like that in Figure 8.22.

Say there is a new request to allocate 1.2 MB. We must first find a chunk large enough to satisfy that request. The 1 MB chunk will not do, but the 2 MB chunk will. But before allocating it, we break that chunk into two buddies, or two 1 MB chunks, and ask whether one of those chunks will do. The answer is no, and so we go back to the 2 MB chunk and allocate it. Call the allocated chunk B. Our memory now looks like that in Figure 8.23.

Say there is a new request to allocate 1.5 MB. We must first find a chunk large enough to satisfy that request. The 1 MB chunk will not do, but the 4 MB chunk will. But before allocating it, we break that chunk into two buddies, or two 2 MB chunks, and ask whether one of those chunks will do. The answer is yes, but rather than allocating the 2 MB chunk, we break that chunk into two buddies, or two 1 MB chunks, ask whether one of those

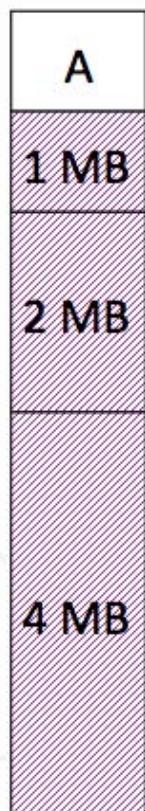


Figure 8.22: One allocated chunk of size 1 MB.



Figure 8.23: A second allocated chunk of size 2 MB.

chunks will do. The answer is no, and so we go back to the 2 MB chunks and allocate one of them. Call the allocated chunk C. Our memory now looks like that in Figure 8.24.



Figure 8.24: A third allocated chunk of 2 MB.

Say there is now a request to free chunk B. We mark chunk B free, and check whether its buddy is also free. The buddy of B is the 2 MB memory area above it, which we see is partially allocated to chunk A. Consequently, the buddy of B is not free, and so there is no coalescing, and we stop. Notice that there are two free chunks that are adjacent, but because they are not

buddies, we cannot coalesce them, and thus consider the new chunk to be of size 3 MB. Rather, we are stuck with two free chunks of size 1 MB and 2 MB. Our memory now looks like that in Figure 8.25.



Figure 8.25: The memory after freeing chunk B.

Say there is a request to free chunk A. We mark chunk A free, and check whether its buddy is also free. The buddy of A is the 1 MB memory area immediately below it, which we see is free. Consequently, since the buddy of A is free, we can coalesce A and its buddy, resulting in a free chunk of 2 MB. But we are not done. We now ask whether the buddy of this 2 MB chunk is

also free, and the answer is yes. So, we coalesce them into a 4 MB chunk. We now ask whether the buddy of this 4 MB chunk is also free, and the answer is no. So, we stop. Our memory now looks like that in Figure 8.26.



Figure 8.26: The memory after freeing chunk A.

To represent the memory in the Buddy System algorithm, with its various allocated and free chunks, we can use a binary tree data structure. Figure 8.27 shows the states of the binary tree during the allocation of chunk A when the memory is free with one 8 MB chunk, after breaking the 8 MB chunk into two 4 MB buddies, and after breaking one of the 4 MB chunks

into two 2 MB buddies.

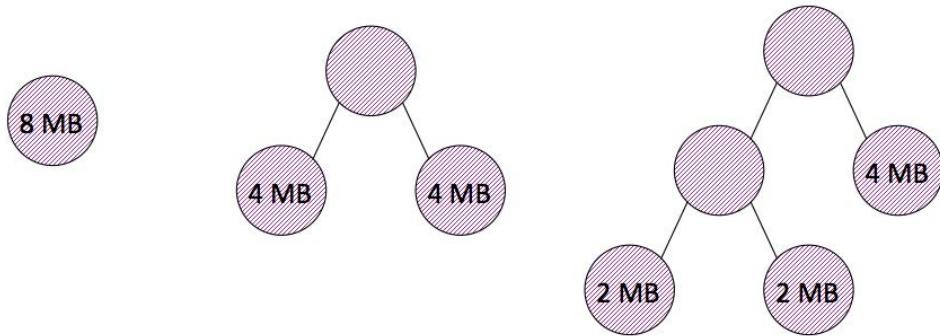


Figure 8.27: The leftmost binary tree, just a single node, represents the memory as one 8 MB chunk. The center binary tree shows the breaking of the 8 MB chunk into two 4 MB buddies. The right binary tree shows the breaking of one of the 4 MB chunks into two 2 MB buddies..

Figure 8.28 shows the state of the binary tree after the allocation of chunk A, and Figure 8.29 show the state after the allocation of chunk B.

Figure 8.30 shows the state of the binary tree after allocating chunk C. Compare this to Figure 8.24, which shows the memory state at this point.

Figure 8.31 shows the state of the binary tree after freeing chunk B. Compare this to Figure 8.25, which shows the memory state at this point.

Finally, Figure 8.32, Figure 8.33, and Figure 8.34 show the sequence of binary tree reductions that occur after freeing chunk A, which results in two coalescing of chunks that are buddies. Compare this to Figure 8.26, which shows the memory state at this point.

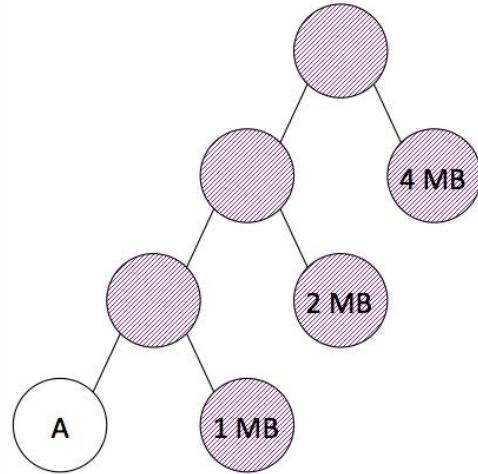


Figure 8.28: After allocating chunk A.

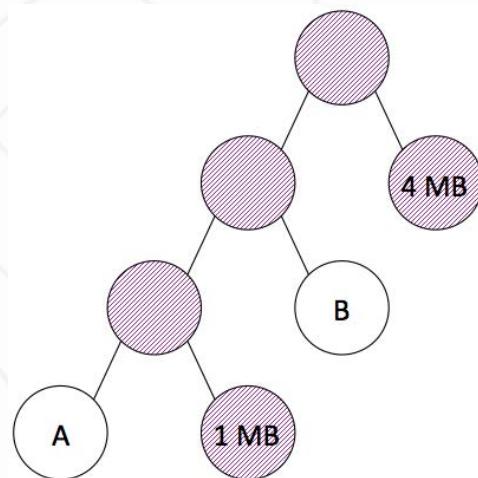


Figure 8.29: After allocating chunk B.

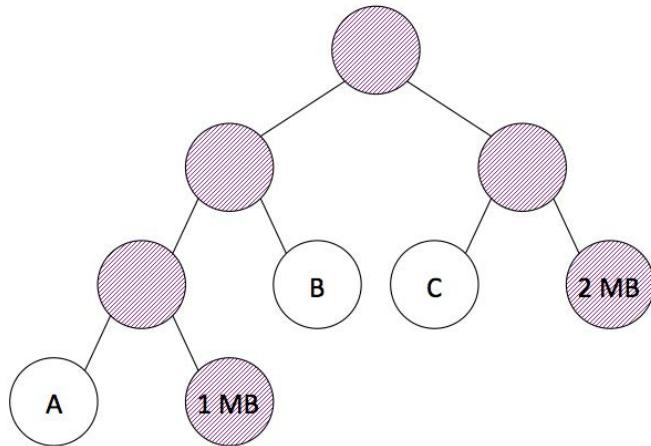


Figure 8.30: After allocating chunk C.

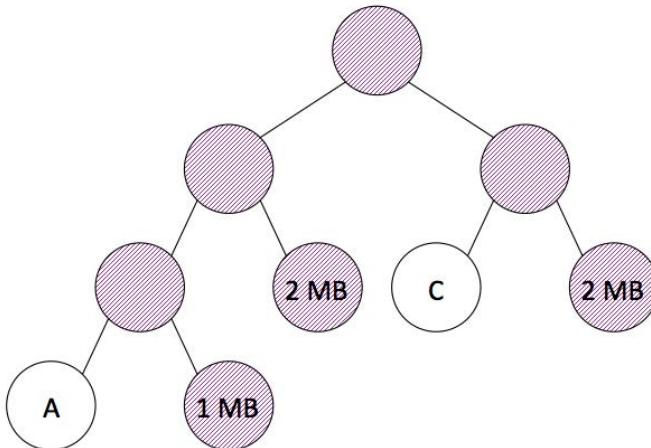


Figure 8.31: After freeing chunk B.

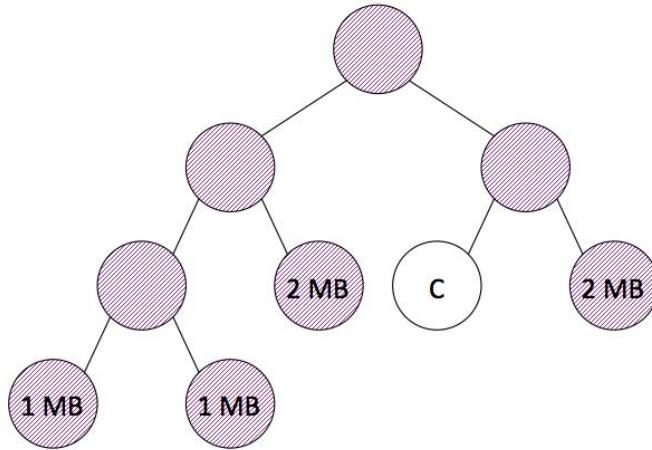


Figure 8.32: After freeing chunk A, but before coalescing any buddies.

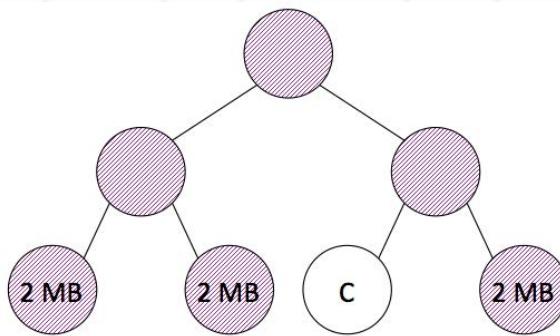


Figure 8.33: After coalescing the 1 MB buddies

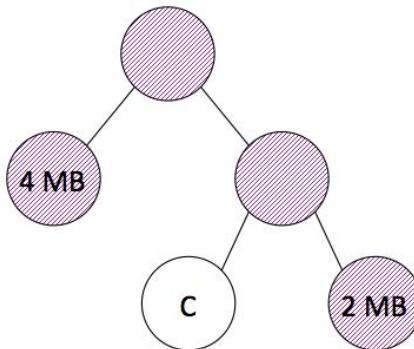


Figure 8.34: After coalescing the 2 MB buddies.

8.17 Summary

Memory management determines how portions of memory are allocated and freed. Examples of memory allocations include the entire memory for a process, and within a process, its areas for text, data and stack. An address space is used to reference a process's memory, and is required by the compiler so that it can generate addresses.

Three algorithms for allocating memory are First Fit (and its variant, Next Fit), Best Fit, and Worst Fit. Of these three, First/Next Fit is preferred because it is faster than the others. Best Fit and Worst Fit involve checking every hole to find the smallest and largest, respectively, hole that will satisfy the memory allocation request, whereas First/Next Fit will stop as soon as a large enough hole is found. A memory will eventually become fragmented. We distinguish between two types of fragmentation, external and internal. External fragments are outside of any allocated blocks of memory, whereas

internal fragments are within allocated blocks.

To deal with not finding any available holes, memory can either be compacted, or we can break requests into smaller ones (so that they will fit the existing holes). The latter is preferred, as compaction takes too much time.

The 50% rule says that, on average, the number of holes will be half the number of allocated blocks. The unused memory rule tells us that the fraction of unused memory (memory contained in holes) will be $k/(k + 2)$, where k is the ratio of average hole size to average block size.

Variable sized blocks lead to fragmentation; consequently, it is worth trying to reduce the variation in the units of allocation. The Buddy System is an effective memory allocation algorithm where memory is allocated in powers-of-2 sized chunks. This is a compromise between memory allocation using a single fixed-size block unit and variable sized blocks.

8.18 Exercises

1. What is meant by “process memory”?
2. What is contained in the Text? Data? Stack?
3. Why should there be different memory areas?**
4. What is an address space?
5. Where are the Text, Data, and Stack located in a process’s address space, and for each, why?**

6. What is the compiler's model of memory?*
7. How does the memory layout in FIGURE XXX relate to the compiler's model of memory?**
8. What are the key aspects of memory NOT known by the compiler, and why are they not known?**
9. How does the goal of supporting multiple processes affect how memory is allocated?**
10. How is this related (if at all) to time-sharing?***
11. What is memory management?
12. What is meant by a “hole”?
13. When a memory region is allocated, how does this affect the number and size of holes?*
14. What is “First fit”? How does it work, and what are its pros and cons?
15. What is “Best fit”? How does it work, and what are its pros and cons?
16. What is “Worst fit”? How does it work, and what are its pros and cons?
17. For each of the above, what effect do variable size memory regions have?***

18. What analysis can be done to determine which memory allocation method is best?**
19. What is fragmentation?
20. What is internal fragmentation?
21. What is external fragmentation?
22. Which type of fragmentation do you think is worse, and why?***
23. When trying to allocate memory of a certain size, what if there is no large enough hole?*
24. What is compaction, and is it a good idea?**
25. What are the pros and cons of breaking up blocks into sub-blocks?*
26. What analysis can be done to determine whether compaction or breaking blocks into sub-blocks is best?**
27. Given a memory with n blocks allocation, what is the minimum number of holes there can be?** the maximum?**
28. What is the 50% Rule?
29. Does the 50% Rule say anything about how much memory is being wasted?*
30. What is the proof for the 50% Rule?***

31. What is the Unused Memory Rule, and what does it say?*
32. Why is k a key parameter for the Unused Memory Rule: what does it mean?*
33. How does the amount of wasted memory vary as a function of k , and why (give an intuitive reason)?**
34. Can you prove the Unused Memory Rule?***
35. What is the value of pre-sizing holes?*
36. What are the pros and cons of pre-sized holes that are all the same size?*
37. What are the pros and cons of pre-sized holes that variable size?*
38. If memory is divided into pre-sized holes, is it more susceptible to internal fragmentation or external fragmentation, and why?**
39. What is the Buddy System?
40. How does the Buddy System work when allocating memory?* when freeing memory?*
41. Is the Buddy System more susceptible to internal fragmentation or external fragmentation, and why?**
42. What is the rationale for the Buddy System in how it allocates and frees memory?***

43. Why is a binary tree a good data structure for implementing the Buddy System?*



Chapter 9

Logical Memory

A *logical memory* is essentially a process's memory. It is memory as viewed and referenced by a process. It is allocated without regard to the physical memory.

Ultimately, processes, and even the kernel, must share the physical memory. There are three problems due to sharing the physical memory that we must deal with:

- The *addressing* problem: the compiler must generate memory references, despite that it lacks information as to where the process will be located in physical memory. And yet, those processes must ultimately run in the physical memory and be able to address it.
- The *protection* problem: given that all processes are in the physical memory, how can we protect them from accessing each other's memory areas.

- The *space* problem: the more processes there are, the less physical memory each can have if they are all to be present in physical memory. And yet, we want to provide them with as much memory as possible.

9.1 Logical vs. Physical Address Spaces

An *address space* is a set of addresses for a memory. It is generally linear, going from 0 to N-1, where N is the number of memory elements that can be addressed (i.e., bytes or words). A physical memory (PM) will have an address space, which we call the *physical address space* (PAS). Typically, the kernel will occupy the lowest addresses, as shown in Figure 9.1.

We distinguish between *logical addressing* and *physical addressing*. As shown in Figure 9.2, there can be many logical memories (there will be one per process), each with its own logical address space, but there is only one physical memory. We can imagine that each process's logical memory will fit somewhere in the physical memory (for the moment, we'll assume there is enough room to do so).

9.2 Mapping Logical Address to Physical Addresses

This requires a logical memory address space to be *mapped* to a portion of the physical memory address space. Logical addresses, which assume a separate

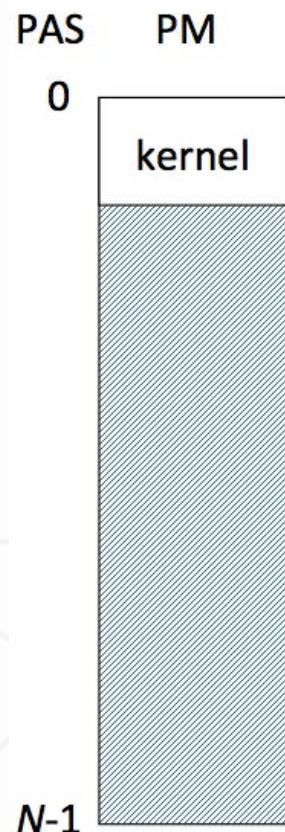


Figure 9.1: A physical memory with its physical address space. The kernel will typically be located at the very beginning.

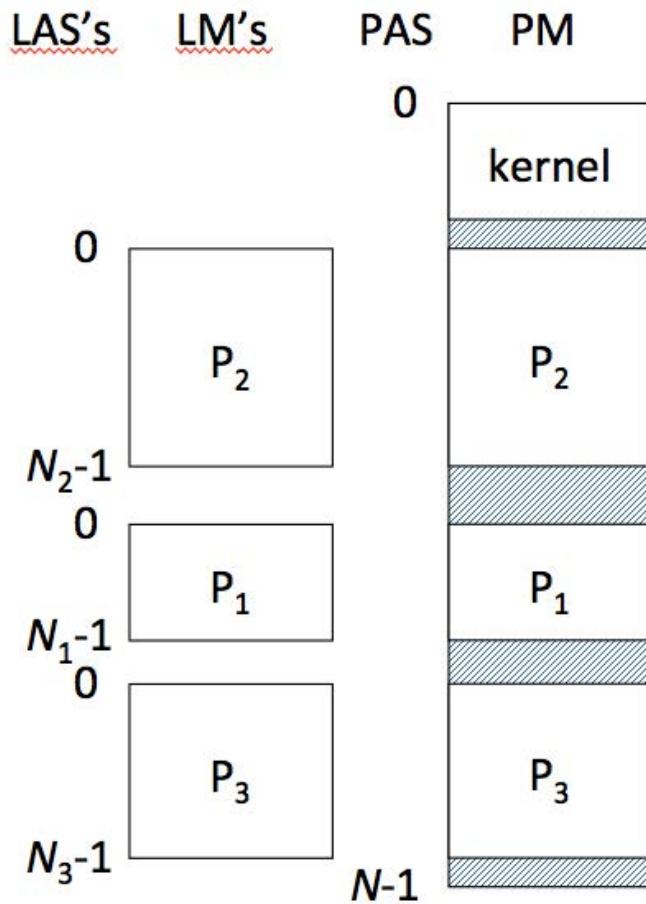


Figure 9.2: Multiple logical memories (LM), one per process, each with their own logical address space (LAS), that are located in the physical memory (PM), with its own physical address space (PAS).

logical memory starting at address 0, are compiler generated. They are independent of the location of processes in physical memory. To implement the mapping, a process's logical address must ultimately be converted into a physical address.

This can be done in two ways: via software, at load time (when the program is loaded into memory when it is to start running); and via hardware, at run time (when the memory is actually accessed). To convert logical addresses to physical addresses at load time, it must be known where the logical memory is located in physical memory. Knowing this, the machine code (generated by the compiler) would be inspected (by a loader program), and wherever there is a logical address, it is converted to a physical address using knowledge as to where the logical memory is to be located in the physical memory.

However, this presents a problem. Not all logical addresses “exist” at load time. Some programming languages, such as C, allow logical addresses, known in C as “pointers,” to be created on demand. A program can “cast” an integer into a pointer, and can even do “pointer arithmetic” by taking two pointers and, for example, add them. Since the resulting pointer, or logical address, only comes into existence at that point in time, i.e., during run time, there is no opportunity at load time to convert it. Consequently, the software solution to converting logical to physical addresses is limited.

9.3 Base and Bound Registers

The hardware approach to converting logical to physical addresses involves hardware support, and is shown in Figure 9.3. The simplest hardware is the inclusion of a register, called the *base register*, which can be loaded with the process's location in physical memory, specifically its lowest address, i.e., the physical address that corresponds to logical address 0. The mechanism works as follows: whenever an instruction is decoded that contains a memory reference, this reference, which is a *logical address* (after all, that's is what the compiler generated as there was no conversion during load time) is added (automatically by the hardware) to the contents of the base register, which is a logical-to-physical address conversion, and that address is what is submitted to the physical memory. To “move” a process in the physical memory, it would be copied in its entirety to the new area, and the base register would be changed to the new start address.

Consequently, with the addition of a simple register, we have essentially solved the addressing problem (though we will want to do even more, as we will see). A compiler can now generate addresses for a hypothetical memory of size N , where N is the size of the process as determined by inspecting the program. This hypothetical memory is what we called the logical memory, and when the program is run, a hole of size N must be found in the physical memory, using one of the memory allocation algorithms discussed in the chapter on memory management. Wherever that hole is, the memory is

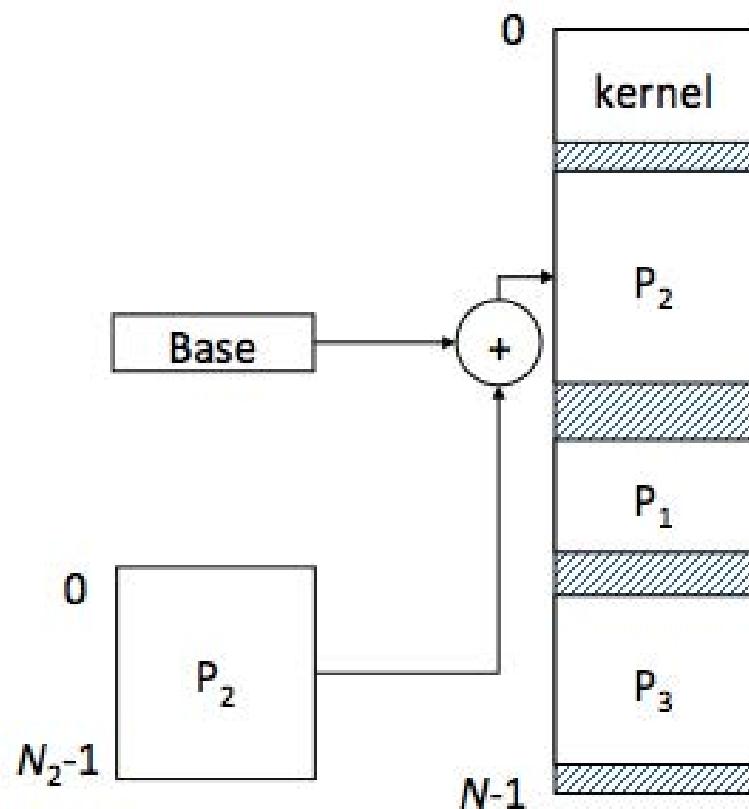


Figure 9.3: Logical-to-physical memory conversion in hardware, using a base register.

allocated for the process, the program is loaded into that memory area, the base register is loaded with the start of that start address of that allocated memory, and now the process can be run.

The same idea can be used to address the protection problem. In addition to the base register, a *bound register* can be added, that will be used to check whether the logical addresses is less than N , the size of the process. This is shown in Figure 9.4. Consequently, when the process is set to run, both the base register is set to the start address in physical memory, and the bound register is set to N . When the process executes, in addition to the base register being added to the logical address, the logical address is also compared to the bound register; the hardware does these two operations in parallel. If the logical address is not less than N , a hardware exception is generated, which, just like executing the TRAP instruction, causes the kernel to run. The kernel will then resolve the bad memory reference, likely terminating the process. Otherwise, if the check passes, the process will proceed (the hardware exception will not have occurred and so the kernel will not be invoked). This happens on every memory reference.

The base and bound form part of the *hardware memory context* of a process, just like the PC and SP registers form part of the hardware CPU context, and just like the latter, it becomes part of the context that needs to be saved and restored on each context switch. One problem that arises is that the code for context switching resides in the kernel, and the kernel has its own location in physical memory.

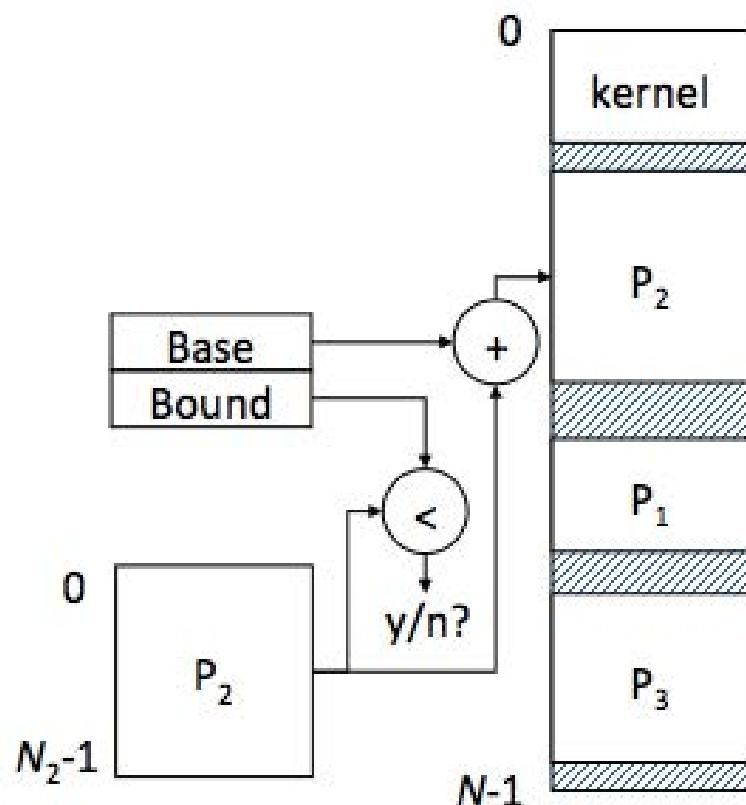


Figure 9.4: Adding a bound register to check for valid logical addresses.

Consequently, when the kernel runs, the base register must have the kernel's start address in physical memory. But how does this get loaded when jumping into the kernel? Furthermore, how does the kernel load the base register with the start address of the next process to run, as once it loads this register, the memory accessed is no longer that of the kernel but that of the next process. We could try to play the same trick we did for the PC register, by making it the last instruction, but the loading the PC can't also be the last instruction. And even if we were able to solve this, there's still the problem of how does the base register get set to the kernel in the first place?

This can be solved as follows. Recall that we placed the kernel at location 0 of physical memory, so that its logical addresses equal its physical addresses. Consequently, the base register would have to get loaded with 0 to properly translate kernel logical addresses. Another approach is to turn off logical-to-physical addressing! This way, the base register is unused, and since we (conveniently) placed the kernel at physical location 0, *no translation is needed.*

And so the solution is as follows. When the kernel is entered such that the machine goes into kernel mode, logical-to-physical addressing is turned off (automatically by the hardware), allowing the kernel to be able to access its memory, and to load the base register with it being affected. Another approach is to have a dedicated base register for the kernel that only operates when in kernel mode, and thus access kernel space, again allowing the kernel

to run and modify the base register when processes run in user space. Either approach will work, the second one being more general, but the first will work with minimal hardware modification. While we as operating system implementers cannot dictate how the hardware works, we are generally given one of these two options, and use them appropriately.

Note that another benefit of placing the kernel at location 0 is that its allocated memory is *before* that of any process. By doing so, a process cannot possibly access the kernel, because the smallest logical address it can express is 0, and its start address in the base register will always be added to it, generating a physical address that is always beyond the memory of the kernel. Of course, given the use of the bound register, a process would not be able to access any memory outside its physical allocation, so getting the protection benefit by placing the kernel at physical location 0 is moot. And yet, it is relieving to know that the kernel is protected, independent of the bound register. What if the bound register value were to get corrupted somehow, or what if we happen to have a machine without a bound register. In either case, our trick of having placed the kernel at physical location 0 will still work.

9.4 Allocating Logical Memory Partitions

We now have a working strategy for being able to flexibly locate and robustly protect the logical memories of processes in physical memory. However, we

have been assuming all along that a process's logical memory will fit in one of the available holes, i.e., free contiguous areas, of physical memory. The obvious problem is that, the larger the logical memory, the more difficult it will be to find a large-enough hole. There may simply be no such hole, as shown in Figure 9.5. We can try to solve this by dictating that the logical memory be small, but is this a reasonable approach? Definitely not.

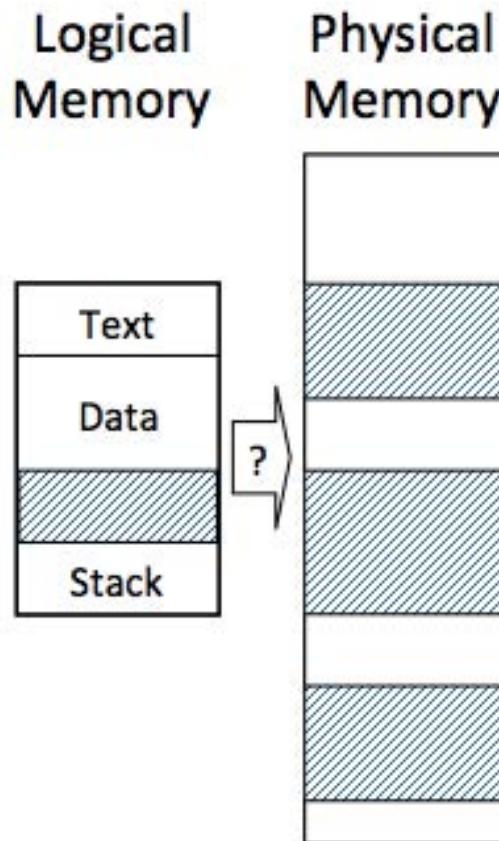


Figure 9.5: A process's logical memory may not fit in any available hole.

There is a good reason for why we want a process's logical memory to

be as large as possible, as it affects the sizes of programs that the compiler is able to successfully compile, and this affects the sizes of data structures the programmer can declare in their programs. With a very limited *logical* memory, if the programmer declares, say, a very large array, the compiler may say that the program cannot be compiled. This would force the programmer to reduce these data structures, and copy data that cannot be stored in them to files, shuffling the data between files and data structures as needed.

This added complexity is exactly what we want to avoid: the operating system should be providing illusions that, just like there are many CPUs, one per running program, there is plenty of memory; hence, the desire to have a support a large logical memory. But, the larger the logical memory, the less likelihood there will be a large enough hole to fit it.

Recall that one of our strategies for memory management, in dealing with the problem of finding a large enough hole to satisfy a memory request, is to break up the request into smaller pieces, with the idea that each of the smaller pieces will then fit in the existing holes. Can we apply this idea to an allocation that corresponds to the logical memory of a process? Recall the partitioning of a process's logical memory into various areas, as shown in Figure 9.6.

There are three distinct areas:

- **Text:** contains program instructions; is fixed size; permissible operations are that contents, i.e., instructions, must be executable

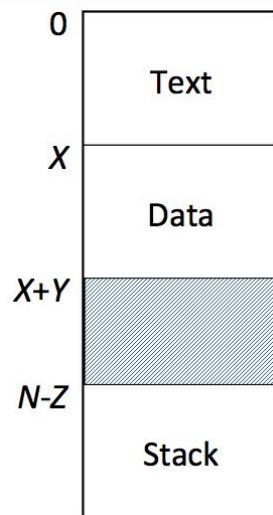


Figure 9.6: The areas of a process's logical memory.

- **Data:** contains variables, such as global/static, and heap (program-directed dynamic allocation); variable size: can grow (due to dynamic allocation); permissible operations include reading and writing of variables
- **Stack:** contains activation records that are automatically allocated/freed upon procedure calls/returns, each of which contain local variables and other per procedure-call state; variable size: can grow and shrink; permissible operations include reading and writing of variables and other call state.

Note that there is also an area between the data and stack that is currently not used, but may be used as either the data or stack area grow. This is part of a process's logical memory, and would have to be allocated as part of any

request to allocate physical memory for the entire logical memory, despite that the memory is actually not used except to allow the data and stack areas to grow. Allocating physical memory for this purpose is clearly inefficient, given that it won't be touched. It is simply an "internal fragment," which cannot be allocated by the operating system for any other purpose other than its potential as future memory for data and stack.

This partitioning into text, data, and stack, suggests a natural breakdown for requesting physical memory for a process. Rather than a single request for an amount for the entire logical memory, we can make three separate requests: one for the text area, one for the data area, and one for the stack area. Each of these will be much smaller than a single request for the entire logical memory. This approach has the added advantage that there is no waste due to internal fragmentation. Consequently, the problem of finding large-enough holes becomes much more promising: we simply need to find large-enough holes for each of the individual memory areas, rather the one large hole for their combined area that also includes an internal fragment to allow for growth, as shown in Figure 9.7.

9.5 Any-Sized vs. Same-Sized Allocation

Note that the idea of breaking up a memory request for the entire logical memory into the smaller ones of the text, data, and stack areas is an instance of the more general idea of breaking up a single large request into many

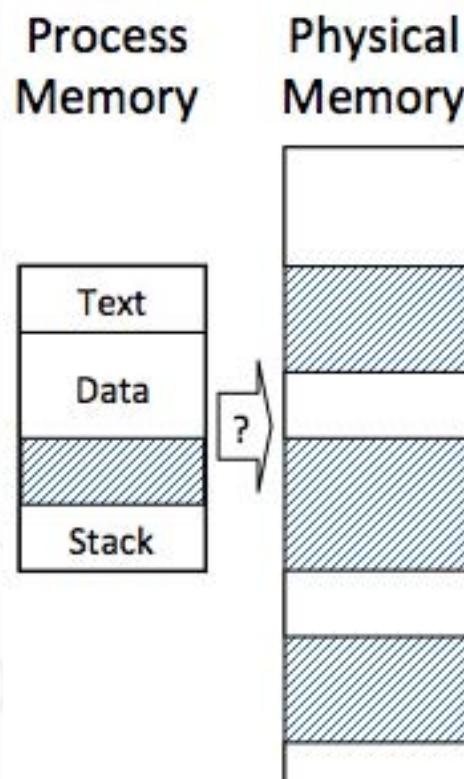


Figure 9.7: Rather than finding a large-enough hole for the logical memory as a single entity, find holes for the text, data, and stack memory areas separately.

smaller ones. How might we support this general idea?

There are two basic opposing approaches, as shown in Figure 9.8:

- Any-size allocation: allow for requests whose sizes can be of any amount
- Same-size allocation: all requests are for memory whose sizes are the same, fixed to some given amount, e.g., 1 KB

In the first approach, a request can be any size. If a memory area of 1375234 bytes is needed, a request of exactly that size would be made, and as long as there is a hole of that size or more available, a portion of the hole would be allocated to satisfy precisely that amount of memory. This will of course leave a smaller hole (barring the rare case of a perfect fit), and over time, the memory will likely become highly fragmented. This is external fragmentation, and is a key characteristic of the any-size allocation approach.

In the second approach, a request must be of a given size. If the given size is, say, 1 KB, only that size of memory can be requested. If a memory area of 246 bytes is needed, the request will be for 1 KB; it cannot be smaller. The 1 KB allocation will be more than enough for the needed 246 bytes, with the remaining 778 going unused. This is internal fragmentation, and is a key characteristic of the same-sized allocation approach.

Let us return to the specific problem of fitting a process's logical memory into the physical memory. Breaking the logical memory up into parts so that

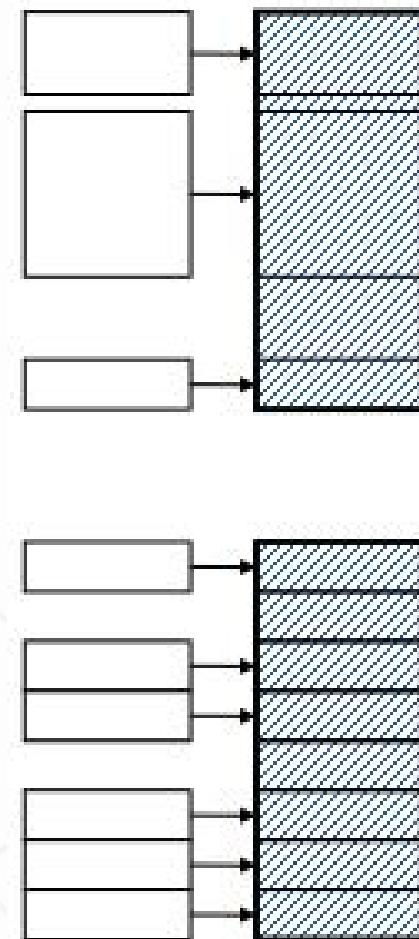


Figure 9.8: Any-size (upper diagram) vs. same-size (lower diagram) allocation approaches.

physical memory requests are smaller, and thus more likely to be satisfied, is a good idea. But, it raises a problem that we have sidestepped.

The logical memory has an address space, which we've assume to start at 0 and increase linearly. When placed at some physical address, we were able to implement address translation (from logical to physical) with the addition of a base register. This simple mechanism will no longer hold, as we are now breaking up the logical memory into pieces, each of which can be placed in different parts of the physical memory.

Looking at Figure 9.7, if the text area has logical addresses 0 to $X - 1$, and it is placed at physical address p , its physical addresses will be p to $p + X - 1$. Given that the data area comes immediately after the text area in the logical address space, its first logical address is X . But, its physical address will *not* be $p + X$ if the data area is placed in another area of physical memory. We need a more advanced hardware mechanism than a base register.

Indeed, there are two basic mechanisms that modern hardware memory management units offer:

- Segmented addressing: the logical memory is organized as a set of memory *segments*, where each segment can be of any/variable size as specified by the operating system (on behalf of a process)
- Paged addressing: the logical memory is organized as a set of memory *pages*, where each page is the same/fixed size as specified by the hardware

Beyond these basic mechanisms, there are more advanced ones, including multi-level paged, hybrid segmented/paged, inverted paged, etc. We will focus on the basic ones, and the hybrid segmented/paged.

9.6 Segmented Memory

A segmented memory is a logical memory composed of a set of *segments*. A segment is a memory that contains typically logically related information, i.e., information that is related in some way, or for which it makes sense to package together, such as all of a program's instructions or its long-lived data variables. A process's text, data, and stack areas (of the logical memory) would be placed in their own text, data, and stack segments, respectively.

Each segment has an identifier s , generally an integer from 0 to $S - 1$, where S is the maximum number of segments (determined by hardware), and can be any specific size (up to some hardware-determined limit), N_s (N_s is the specific size of segment s). Segment addresses are of the form (s, i) , where s is the segment number, and i is the offset within segment s . Note that i must be between 0 and $N_s - 1$.¹

Segments can also be protected with permissions, which indicate which

¹We are assuming that the unit being used to measure segment sizes is also the unit of memory that is addressable (we made the same assumption earlier when discussing memory, in general). For example, if sizes are given in terms of bytes and the memory is byte addressable (a common situation), then a segment of size N has addresses that range from 0 to $N - 1$. But it doesn't have to be this way. For example, memory might be measured in bytes, but may be word addressable, where each word is 4 bytes. In this case, a segment of size N would have addresses from 0 to $\text{INT}(N/4)+1$, where $\text{INT}(x)$ is the integer part of x , or the FLOOR of x .

operations can be applied to their contents. The most common set of possible permissions are read (indicated by “r”), write (indicated by “w”), and execute (indicated by “x”). The specific permissions for a segment are expressed as a 3-character code, such as “rwx” which means all three permissions are present. A dash “-” in place of any of the three characters means that that permission is missing. For example, “r-” (read-only) means that read permission is present, but write and execute permissions are not. A text segment will commonly have permissions set to “-x” (execute-only), while a data segment would have “rw-” (read and write, but not execute) permissions. This is in keeping with the idea that a segment contains information of the same type, as such information would lend itself to the same type of operations.

Figure 9.9 shows a segmented memory consisting of four segments, numbered 0 through 3.

Segment 0 is a text segment containing code, has size N_0 , and has addresses ranging from 0 to $N_0 - 1$. The segment has permissions of “-x” (execute-only).

Segment 1 is a data segment containing data variables, has size N_1 , has addresses ranging from 0 to $N_1 - 1$, and has allowed permissions of “rw-” (read and write, but not execute). It can also grow or shrink in size. In fact, all segments can in theory have their size changed, but some segments, such as text segments, are expected to remain fixed given their initial size.

Segment 2 is a stack segment containing activation records, has size N_2 , has addresses ranging from 0 to $N_2 - 1$, and has allowed permissions of “rw-”

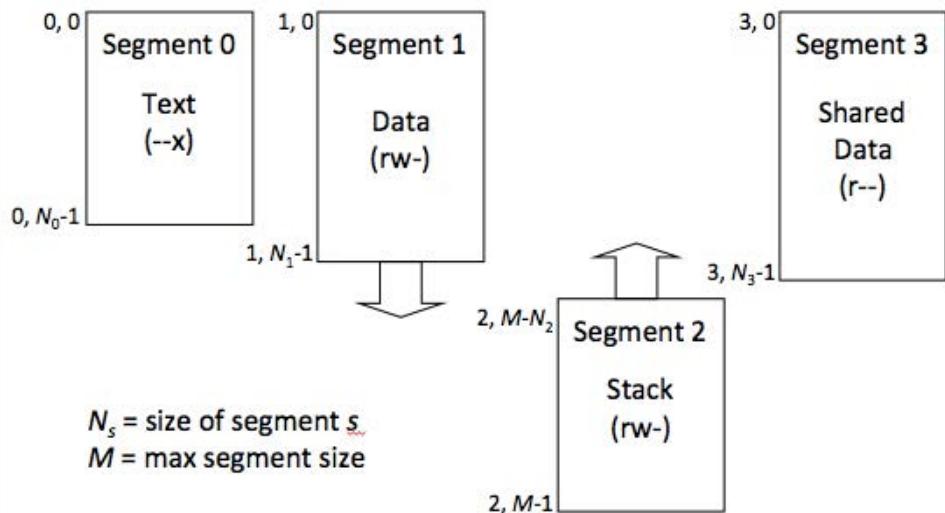


Figure 9.9: A logical memory consisting of four segments.

(read and write, but not execute). It can also grow or shrink in size, but does so in the opposite direction than a data segment, i.e., it occupies the highest addresses, from $M - 1$ to $M - N_2$, where M is the maximum possible size of a segment, and grows towards smaller addresses.

Finally, segment 3 is a data segment, has size N_3 , has addresses ranging from 0 to $N_3 - 1$, and has allowed permissions of “r-” (read-only). It is not expected to change in size. Segment 3 is labeled as “shared data,” meaning that other processes can also access it.

Consequently, from segment 3 we see another characteristic of segments; processes can *share* segments (and share them *selectively*, i.e., a segment can be shared or not be shared, and if shared, by a specific set of processes), which means that they will appear in the logical memories of other processes.

If one process modifies that segment, all the other processes that are sharing that segment will see that modification.

Typically, while each process may have a different ID for a shared segment (e.g., process 1 might refer to it as segment 2, while process 2 might refer to the same shared segment as segment 4), the memory being referred to is the same, i.e., there will be a *single copy* of the segment in physical memory, even though each represents a segment of *different* logical memories.

With a segmented logical memory, a compiler has the flexibility of determining the number of segments a process will need – generally at least three: text, data, and stack – and can then generate addresses for each without concern for where they are located *relative to each other*. As far as the compiler is concerned, each segment is an independent logical memory, can be separately protected by specifying its size and by setting permissions that make sense for that segment, and can even grow or shrink (without concern of “bumping” into another segment). This abstraction of an independent separately addressed and protected logical memory, which can grow/shrink as needed, of which there can be as many as needed by a process, and which can even be shared between processes, is both very powerful and convenient, significantly simplifying code generation (and specifically, memory allocation) by the compiler.

However, ultimately, the segments as determined by the compiler must be placed in the physical memory (using some memory allocation algorithm, such as First Fit or The Buddy System, as discussed in Chapter 8), and there

must be a mechanism for translating a segment address of the form (s, i) into a physical address a , as shown in Figure 9.10.

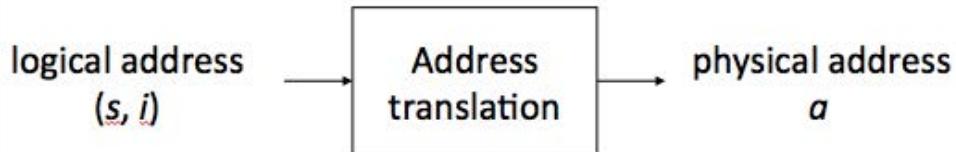


Figure 9.10: The address translation mechanism must convert a logical address of the form (s, i) into a physical address a .

When we considered a single monolithic logical memory, the translation of a logical address into a physical address was done using a base register, which contains the start address of the logical memory in physical memory, and which is added to each logical address to determine the physical address, as shown in Figure 9.3 and Figure 9.4. However, now that we have multiple segments, each of which can be separately located in physical memory, we would need many such base registers, one per segment. Having many base registers in hardware, and having to save and restore them on every context switch, can be expensive. Consequently, modern hardware memory systems support segmented logical-to-physical address translation through the use of in-memory segment tables.

A *segment table* is a typically *per-process* table, i.e., an array of entries, where each entry represents a different segment. The segment table ST effectively implements a function, $ST(s) = b$, which, given a segment number s , produces a physical address b , which is used as the base address (in physical

memory) for that segment. This is then added to the offset i to produce the physical address a for the logical address (s, i) :

$$a = ST(s) + i \quad (9.1)$$

As segments are identified by an integer s , which can range from 0 to $S - 1$ (S is the maximum number of segments), the $s + 1^{st}$ entry corresponds to segment s , e.g., 1st entry corresponds to segment 0, the 2nd to segment 1, etc. The total number of entries can be up to S , but will generally be much less (as S is typically a very large number, much larger than what processes will typically need, thus creating the illusion that a process has an unlimited number of segments at its disposal).

For example, the hardware-supported *maximumpossible* number of segments might be $2^8 = 256$, or even $2^{16} = 65536$, even though the typical process will only require 3 (for text, data, and stack), and may create a few more on demand, making a typical segment table have a maximum of 4 to 8, or even 16 in unusual cases, entries (usually a power-of-2, for reasons that will become evident), the latter being the maximum number of segments a process can create, and is set by the operating system. Obviously, this number cannot exceed the hardware-supported maximum.

A process's segment table is located in the kernel (all segment tables are managed by the kernel), specifically in the kernel's data area, and thus will be in physical memory, as opposed to being in the form of hardware registers.

There are, however, two hardware registers that are required to support a segmented memory, and they are the segment table base register (STBR) and the segment table size register (STSР), as shown in Figure 9.11.

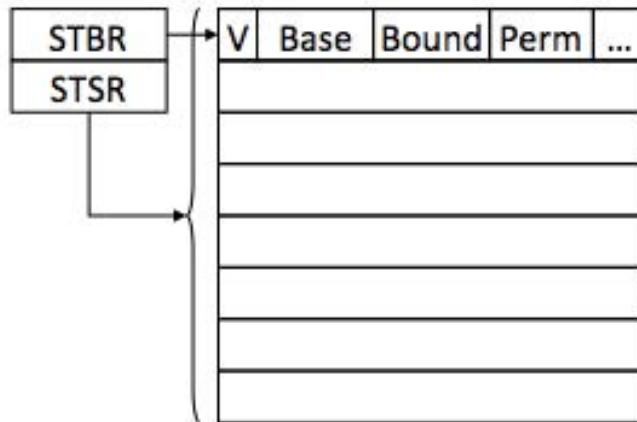


Figure 9.11: A segment table, whose location in physical memory is indicated by the STBR, its size in entries is indicated by the STSR.

The STBR points to the start of the current process's (the one that is running) segment table, and the STSR indicates how big it is, i.e., how many entries it has. The STBR register basically tells the hardware where to find address translation information that will be needed for every memory access to allow it to automatically do logical-to-physical address translation. The STSR register tells the hardware what part of the physical memory corresponds to segment table information, thus preventing it from using any memory beyond it.

Each segment table entry contains various fields:

- **V:** the valid bit, which if 1, indicates that the entry is valid and thus can

be used for address translation, and if 0, indicates that the information is invalid and should not be used

- **Base:** the base address in physical memory of the segment
- **Bound:** the size of the segment
- **Perm:** permissions bits, typically three for read, write, and execute

We are presenting what a typical segment table looks like, but ultimately, the precise format of a segment table is hardware specific. This might include additional information stored in each entry, but that we need not discuss here, as the fields we present are the essential ones to understand how address translation works for a segmented logical memory.

Before proceeding further, it is important to distinguish between the roles of the kernel and the (memory management) hardware. It is up to the kernel to configure the segment table so that the hardware can then make use of it on each memory access. Consequently, when a process is created, the kernel determines the memory needs of the process, provided by the compiler as directives. These would include the number of segments needed (e.g., text, data, and stack), and their initial sizes. The kernel would then create a segment table (in its memory), and make it large enough so that the number of entries is at least the number of needed segments. In fact, the kernel will generally make the table large enough to have enough entries for both the initial segments as well as additional entries for segments that may be created on demand during the execution of the process.

The kernel would then look for holes (e.g., via First Fit) in physical memory so that space for the initial segments can be allocated. In doing this, the kernel will determine the physical addresses of where the segments are located. It then records these addresses in the base address fields of the respective segment entries in the process's segment table. It also records the sizes of the segments in the bound fields. Depending on the type of each segment, it will set the permissions fields according to the type of allowable operations. And finally, it sets the valid bit for each entry that was just configured, indicating that those entries have valid segment information.

For example, say that the program file generated by the compiler indicates that the process is to have three segments, as follows:

- A text segment of size 101384 bytes
- A data segment of size 218894 bytes
- A stack segment of size 100000 (despite that the stack is empty, it is given some initial size with the expectation that it will grow)

Given this, the kernel will create a segment table for the process with, say, eight entries (typically a kernel-configured constant), three for the given segments, and five for ones that might be created during the process's execution. The kernel will then check the physical memory for three holes that are large enough to contain these segments, with some preset size for the stack given that it will be expected to grow when the process runs.

Say that kernel finds three holes with the following address ranges (237352–574232), (501002–620174), (817872–955552). Given the first is large enough to store the data segment, it allocates memory at the address range (237352–456246) (leaving a smaller hole at (456247–574232)). The second can be used for the text segment, allocating memory at the address range (501002–602386) (leaving a smaller hole at (602387–620174)). The third can be used for the stack, allocating the memory at the address range (855552–955552) (leaving a smaller hole at (817872–855551)). Notice that the opposite higher-address end of the last hole is allocated for the stack, since it is expected to grow towards lower addresses.

Given these allocations, the kernel will fill the segment table entries as follows:

- Entry 0 (text): base = 501002, bound = 101384, perm = “-x”
- Entry 1 (data): base = 237352, bound = 218894, perm = “rw-”
- Entry 2 (stack): base = 855552, bound = 100000, perm = “rw-”

The valid bits for these entries will be set to 1, while the valid bits for all the other entries will be set to 0. The segment table for the newly created process is now configured.

When the process is to run, there will be a context switch to it. The kernel will set the STBR to the start address of the segment table (which it knows because this is a data structure in the kernel), the STSR will be set

to 8 (since there are eight entries), the CPU registers will be loaded (the SP will be set to the highest address in the stack segment, (2, 99999), and the PC will be set to the first address of the text segment, (0, 0). Upon loading the PC, the process begins running. Notice that the SP and PC use *logical addresses* rather than physical addresses.

Once the process begins running, all memory references, as they appear in the process's memory (e.g., machine instructions with memory address operands, in the text segment, segment 0) will be logical addresses in the form (s, i) . In fact, the logical address appears as a single word of memory, which is partitioned into an upper set of bits that represent the segment number s , and the lower set of bits representing the offset i , as shown in Figure 9.12.

The number of bits for each part (segment and offset) is fixed and determined by the hardware. Since a logical address is stored in a word, say that the word size in bits is w , the segment number portion of the logical address is n bits, and the offset portion is k bits, where $n + k = w$. Then the maximum possible number of segments is 2^n , and the maximum possible segment size is 2^k . For example, for a 32-bit word, partitioned into 10 bits for the segment number and 22 bits for the offset, the maximum number of segments is $2^{10} = 1024$, and the maximum segment size is $2^{22} = 4 \text{ MB}$.

Keep in mind that these are absolute machine limits. The operating system will generally impose its own limit of the maximum to a much smaller number that is reasonable in practice (and which will be the default number

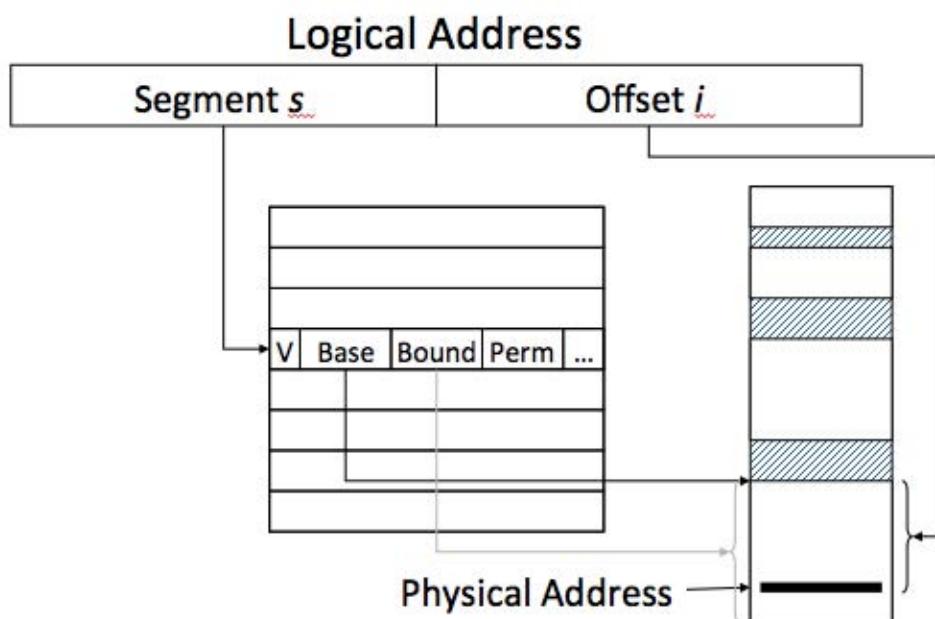


Figure 9.12: A logical address encoded in a word of memory, partitioned into high-order bits representing a segment number s and low-order bits representing offset i .

of entries in the segment table). The reason for this is that the machine design does not presuppose the operating system that will run on it. Some operating systems may make use of many more segments per process than others, and so the machine must provide a maximum adequate for all operating systems expected to run on it.

Say that machine-determined maximum number of segments is 1024 (i.e., $n = 10$). If an operating system, however, generally uses only 3 plus possibly a few more segments per process, it can impose its own limit of, say, 8 segments. This means that segment tables will be limited to 8 entries, rather than 1024 entries if the hardware maximum was used, which is a significant savings. Consequently, only 3 (to encode a number for up to 8 segments) of the 10 bits in the segment number portion of the logical address will be used; the others are essentially wasted.

We now resume our discussion of how address translation occurs. The hardware strips the high-order bits containing the segment number from the word containing the logical address, which it will use it to index into the segment table (which it finds via the STBR). It will check to make sure the segment number s references an entry within the confines of the segment table, and so it tests whether the segment number is less than contents of the STSR, which was previously set with the number of possible entries in the segment table, as shown in Figure 9.13.

If the test fails, the logical address is bad, and cannot be translated; this is an address translation *exception*. How could it happen that the logical ad-

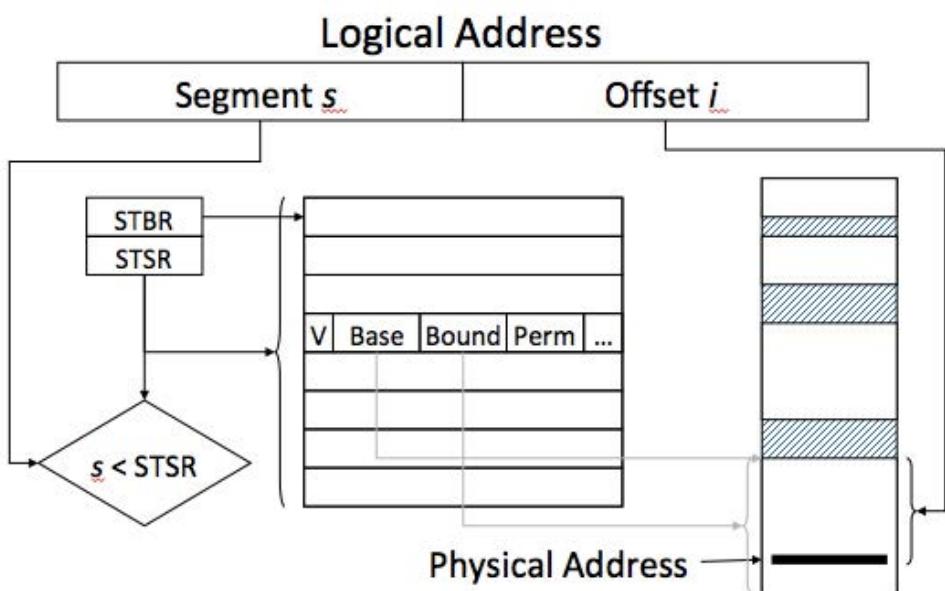


Figure 9.13: The segment number s must be less than the contents of the STSR, which contains the total number of entries in the segment table.

dress is bad, given that the compiler is not expected to produce bad logical addresses? The problem can occur if the program itself generated the bad logical address during its execution, as is possible with the C programming language where a pointer can be created based on turning integers into memory addresses, combining pointers using arithmetic to generate new pointers, etc. A typical resolution to this problem is to trap into the kernel, which then allows the kernel to terminate the process, or to execute an exception handler that was pre-registered by the process with the kernel.

If the test passes, the hardware uses the segment number s to index into the segment table. This is done by multiplying s by the number of bytes per entry, the latter of which is usually a power of 2, and if so, the multiplication can be very efficiently achieved by shifting s to the left by that power of 2 rather than actual multiplying. For example, if each entry required 4 bytes, the segment number s would be shifted to the left by 2. This is then added to the contents of the STBR, with the resulting address being the entry for segment s .

As a hardware optimization, the test that checks whether s is less than the STSR and the calculation to determine the entry address are done in parallel, with the latter being discarded if the test fails. Then, as described above, the failed test would cause an exception that is then handled by the kernel, resulting in the termination of the process or the execution of an exception handler.

Now that the segment's entry has been determined, the hardware inspects

the various fields. The first is the valid bit, which indicates whether the entry has valid information in it, as determined by the kernel. Recall that the kernel had previously filled the fields with information, indicating where the segment is located in physical memory (base), how big it is (bound), and the allowable permissions (perm), and finally sets the valid bit to 1 to indicate the information is valid. Consequently, the hardware checks the valid bit, as shown in Figure 9.14.

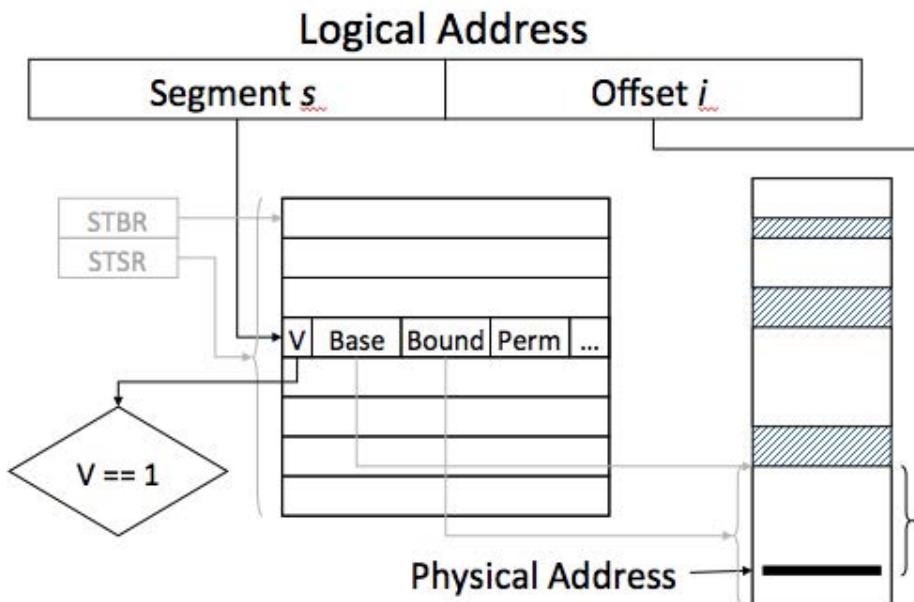


Figure 9.14: Indexing the entry corresponding to the segment, and checking the valid bit.

If the valid bit is 0, it means the kernel did not fill the entry, and so the information cannot be relied on. If this is the case, control goes to the kernel and the kernel will resolve the problem. Why would the kernel not have filled

the information in the entry? We will have to defer this question to the next chapter. Otherwise, if the valid bit is 1, the entry has good information, and the hardware address translation continues.

Next, the offset is tested to see if it is less than the contents of the bound field. This is a check to see whether the logical address is pointing to a location within the segment, or beyond its (upper) bound, as shown in Figure 9.15.

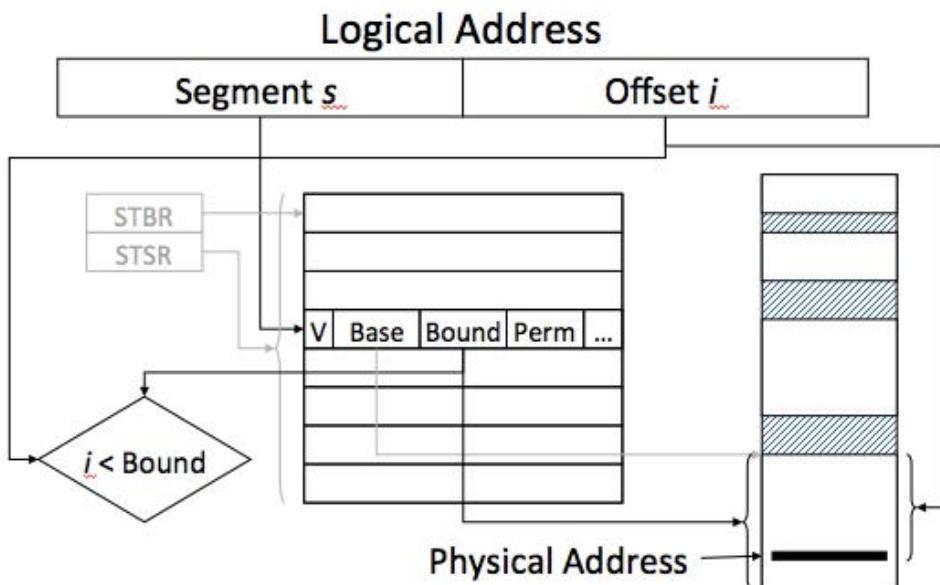


Figure 9.15: Check to see whether the offset is addressing a location within the segment.

If the test fails, an exception is generated, indicating a bad logical address, control goes to the kernel and the kernel resolves as above. If the test succeeds, the offset is less than the bound value, and the hardware address

translation continues.

Next, the operation that is to be performed on the logical address – reading, writing, or executing, its contents – is considered as to whether it is permitted, by checking whether the corresponding permission bit is set to 1, as shown in Figure 9.16.

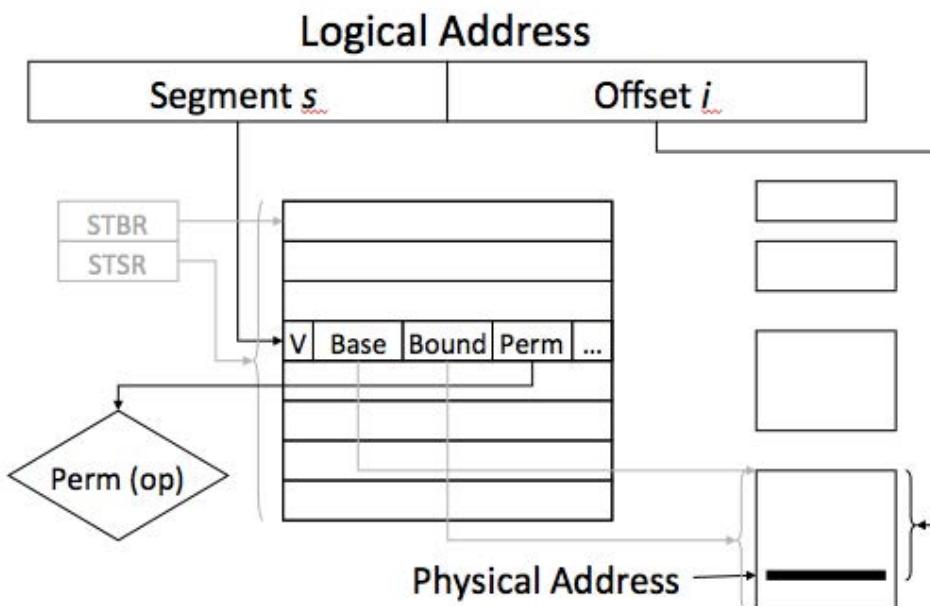


Figure 9.16: Check to see whether the memory operation is permitted.

If the test fails, an exception is generated, indicating that the operation is not permitted (this is a different exception than a bad address), and control goes to the kernel. Such an exception will generally cause the process to be terminated. If the test succeeds, the operation is permitted, and the hardware address translation continues.

Finally, the value in the base field, which indicates the start of the segment

in physical memory, is added to the offset. This is what actually converts the logical address into a physical address, as shown in Figure 9.17.

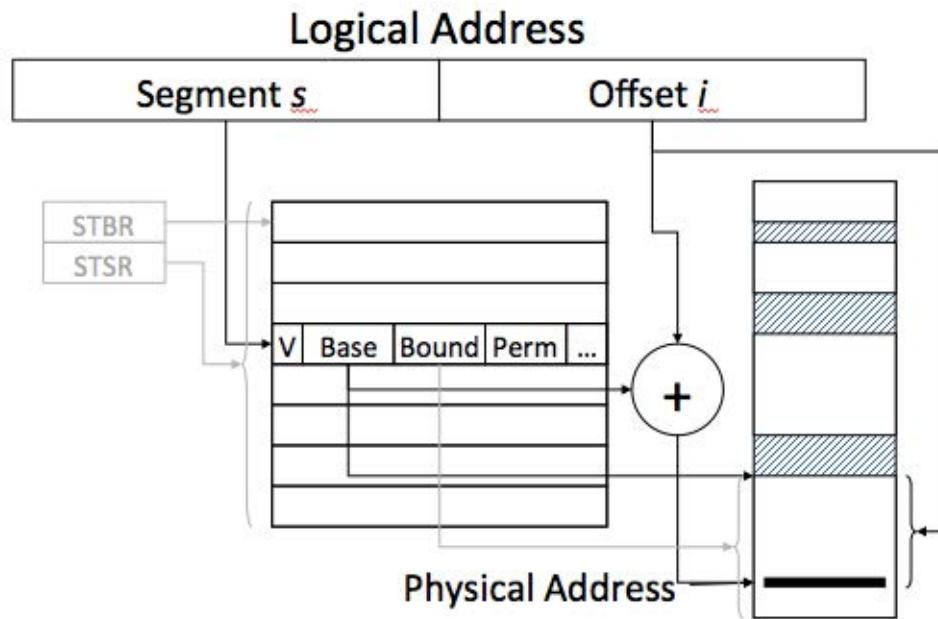


Figure 9.17: The base value and the offset are added to arrive at a physical address.

This address can now be submitted to the physical memory, and the operation that generated this memory reference will be applied.

All of the steps from the indexing into the segment table by adding the value in the STBR to the appropriately shifted segment number, to the various tests, to the determination of the physical address and finally submitting it to the physical memory, will be done by the hardware. This is a significant amount of work that has to occur *for every memory reference!* Consequently, as much as possible is done in parallel; but even so, the overhead of address

translation is non-trivial. We will return to this point later.

How big can a process's segment table be? We can determine its size by calculating the segment table's length and width, and multiplying them. To determine the length and width, we need to be given certain parameters that will be dictated by the hardware: the format of the logical address, specifically the number of bits for the segment number and for the offset; and the maximum size of the physical memory. These parameters will give us enough information to determine the segment table's size, as shown in Figure 9.18.

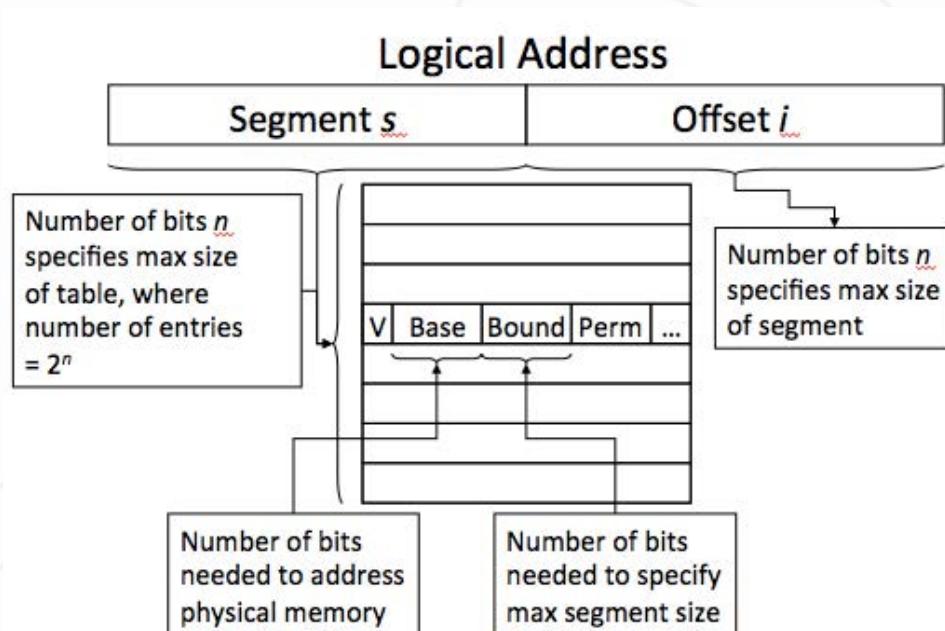


Figure 9.18: Parameters affecting the calculation of the segment table size.

Let's go through the details of how the size is determined based on these

parameters. Given a logical address where the number of bits for the segment number is n , the maximum number of entries is 2^n . This tells us one dimension of the segment table, its length. If we can determine the width, we can calculate the size.

To find the width, we need to determine the number of bits for each field. The valid bit contributes 1 bit. The base value determines where a segment is located in physical memory; consequently, it must have enough bits to address any location in physical memory, and so depends on the maximum size of the physical memory. Assuming a maximum physical memory size of M , the number of bits required to address such a memory is $\log_2 M$.

Next we have the bound field. The bound value determines how big a segment can be; consequently, it must have enough bits to specify the size of the largest possible segment. How do we determine this? We get this by looking at the offset field in the logical address; the number of bits in the offset determines how much can be addressed within a segment. The smallest offset is 0 (all the bits are 0), and the largest offset is $2^k - 1$, where k is the number of bits in the offset field. Thus, the maximum size of a segment is $(2^k - 1) - 0 + 1$ (largest address minus smallest address plus 1), which is 2^k , consequently, k bits are needed to express segment sizes that can range from the smallest to the largest possible segment.

Next we have the permission bits. For this, we will assume 3 bits, to express read, write, and execute permissions.

Finally, we note that a segment table entry should be an integral number

of bytes, and better yet, a power-of-2 number of bytes. To motivate why, imagine that we needed, say 23 bits for all the previous fields. It would be very inconvenient to address the entries of a segment table if the first entry were at byte 0, the next entry began at the last bit of byte 3 (the 24th bit), the next entry began at the second-to-last bit of byte 6 (the 47th bit), etc. So, at the very least, each segment table entry should start on a byte boundary. But better yet, making the number of bits a power of 2 allows for fast indexing. For example, if the number of bits were 64, which corresponds to 8 bytes, then if we want to index to segment table entry j , the starting byte of this entry is simply j shifted left by 3, which is a fast way of multiplying j by 8, the size of each entry. Consequently, we will dedicate p bits for padding such that the total size of the entry is a power-of-2 number of bits.

Summarizing what we have so far for the number of bits for a segment table entry, i.e., the width of the segment table:

- 1 bit for the valid bit
- $\log_2 M$ bits for the base field, where M is the maximum size of physical memory
- k bits for the bound field, where k is the number of bits in the offset field of the logical address
- 3 bits for the permissions field
- p bits for padding to make the width a power-of-2 number of bits

and so, adding all the bits, and dividing by 8 to convert to bytes, the width of the segment table is $(1 + \log_2 M + k + 3 + p)/8$.

Recalling that the maximum length of the segment table is 2^n , where n is the number of bits for the segment number in the logical address, we can multiply this length by the width we just determined to arrive at a formula for sizing the segment table:

$$\text{segment table size} = 2^n \times (1 + \log_2 M + k + 3 + p)/8 \quad (9.2)$$

Let's consider an example. Given the following parameters (determined by hardware):

- A logical address of 32 bits, with 5 bits for the segment number and 27 bits for the offset
- A maximum physical memory size of 1 GB

From these parameters, we can determine the sizes of the various fields:

- 1 bit for the valid bit
- 30 bits for the base field, since $\log_2 1 \text{ GB} = 30$
- 27 bits for the bound field
- 3 bits for the permissions field
- 3 bits for padding, since the total of the above is 61, and adding 3 bits gets us to 64, a power of 2

Consequently, the width of the segment table will be 8 bytes.

The length of the segment table is $2^5 = 32$, since 5 bits are used for the segment number in the logical address. Thus, the size of the segment table in our example is 32×8 ($L \times W$) = 256 bytes. This calculation is summarized in Figure 9.19.

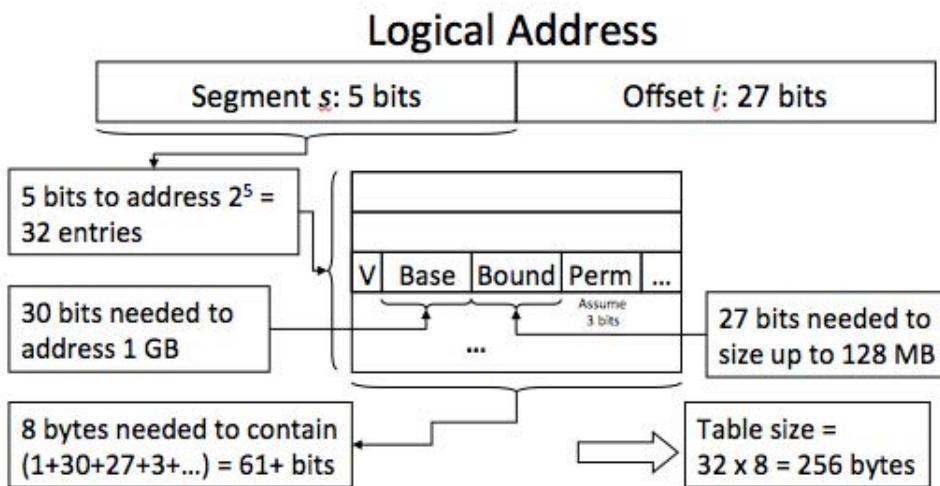


Figure 9.19: An example of sizing the segment table.

We conclude our discussion on a segmented logical memory by reviewing its advantages and disadvantages. The major advantage is that a process can specify use of many (versus one) logical memories, created and organized based on their logical use, according to their purpose. These different uses imply separate packaging, naming, permissions, sizing, and allowing or not allowing changing in size. These properties also make segments amenable to sharing between processes, as the default memory requirement for most

processes is that their memories be private, but if a portion can be configured appropriately (in size, permissions) and there is a need for its specific contents to be accessible to other processes, then being able to create a separate segment is of significant value.

There is also the advantage to the system that smaller memory allocation requests have a greater chance of being successful, i.e., a large-enough hole can be found, than larger requests. In the case of the logical memory of a process, the larger request corresponds to the entire monolithic logical memory, while the smaller requests correspond to its being broken up into segments.

The disadvantage is that, in general, segments can be any size, and so their allocation requests correspond to the any-size approach for allocation. As we discussed above, a key characteristic of the any-size allocation approach is external fragmentation. And as we reviewed in the last chapter, external fragmentation can lead to inefficiencies both in time and space, the time to find a large enough hole (e.g., the time to run First Fit), and the space wasted due to external fragments being so small that they become useless. This motivates the alternative approach to a segmented logical memory, and that is a paged logical memory.

9.7 Paged Memory

A paged memory is a logical memory composed of an array of *pages*. A page is a fixed-sized unit of memory, with all pages being the same size. This size is determined by the hardware, with a typical size being 1 KB (4 KB, 64 KB, even 1 MB, would not be unusual); it must be a power-of-2 number of bytes.

Given a paged logical memory, it is useful to view the physical memory as being composed of an array of *frames*. Like a page, a frame is a fixed-sized unit of memory, with all frames being the same size. In fact, this size is exactly the same as the page size. Consequently, any page of logical memory can be located in any frame of physical memory, as shown in Figure 9.20.

Each page has an identifier p , an integer from 0 to $N_L - 1$, where N_L is the maximum number of pages (determined by hardware) of the logical memory, and has size P (all pages have the same size). Page addresses are of the form (p, i) , where p is the page number, and i is the offset within page p . Note that i must be between 0 and $P - 1$.²

We need a mechanism for translating a page address of the form (p, i) into a physical address. To develop this translation mechanism, we make the following observations. Just as the logical memory is composed of an array of pages, the physical memory is composed of an array of frames. Just as a

²As we did with segments, we are assuming that the unit being used to measure page sizes is also the unit of memory that is addressable. For example, if sizes are given in terms of bytes and the memory is byte addressable (a common situation), then a page of size P has addresses that range from 0 to $P - 1$. But it doesn't have to be this way. For example, memory might be measured in bytes, but may be word addressable, where each word is 4 bytes. In this case, a page of size P would have addresses from 0 to $P/4$.

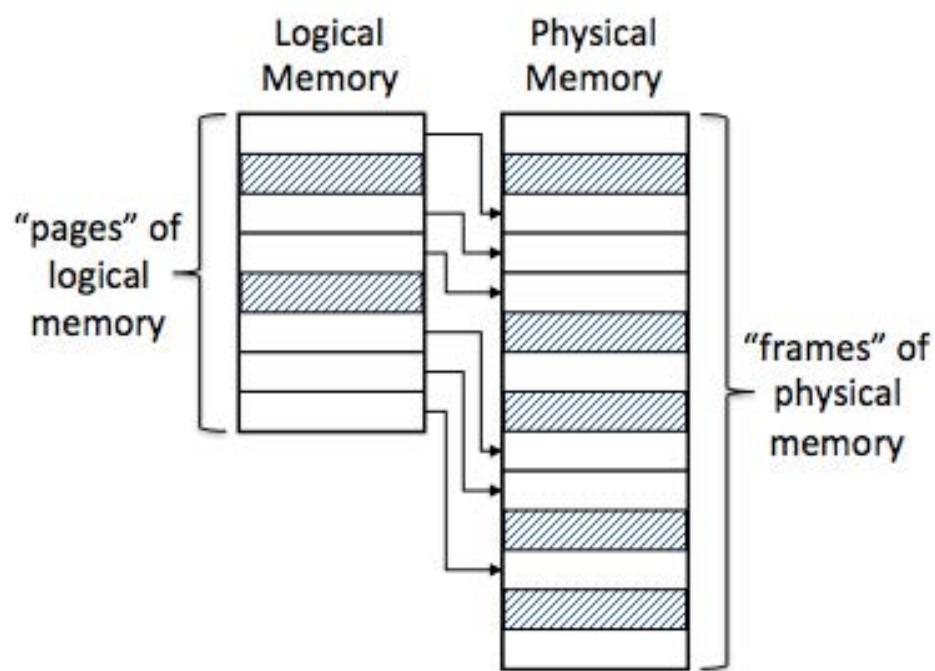


Figure 9.20: A paged logical memory.

logical address is a pair of numbers, (p, i) , where p is the page number and i is the offset within that page, we can interpret the physical address as a pair of numbers, (f, i) , where f is the frame number and i is the offset within that frame. Just as the page number is effectively an index into the array of pages of logical memory, the frame number is an index into the array of frames of physical memory. Finally, observe that since a frame is the same size as a page, the offset i in both the logical address (p, i) and the physical address (f, i) is the same.

These observations lead to the following conclusion: to translate a logical address (p, i) into a physical address (f, i) , we need a function that will translate the page number p into a frame number f , as shown in Figure 9.21.

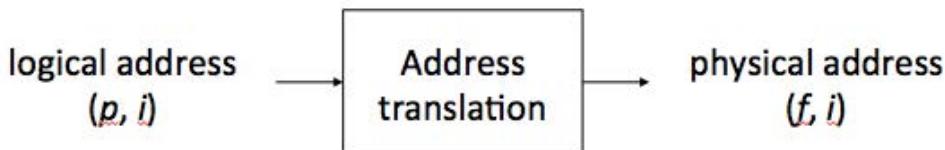


Figure 9.21: The address translation mechanism must convert a logical address of the form (p, i) into a physical address $a = (f, i)$.

The frame number can then be concatenated³ to the offset to form the physical address a :

$$a = f \mid i \quad (9.3)$$

³Concatenating an integer f to an integer i corresponds to taking the bit string that encodes f , including any leading zeroes of f , and appending to it the bit string of i , including any leading zeros of i . For example, if $f = 5$ and $i = 3$, whose bit strings are 000101 and 0011, respectively, then $f \mid i = 0001010011$.

The conversion of a page number into a frame number is achieved using a *page table*, which is a *per-process* table, i.e., an array of entries, one per page. Entries and pages are in one-to-one correspondence such that entry p in the page table corresponds to page p in the array of pages in the logical memory. The page table PT effectively implements a function, $PT(p) = f$, which, given a page number p produces a frame number f , as shown in Figure 9.22.

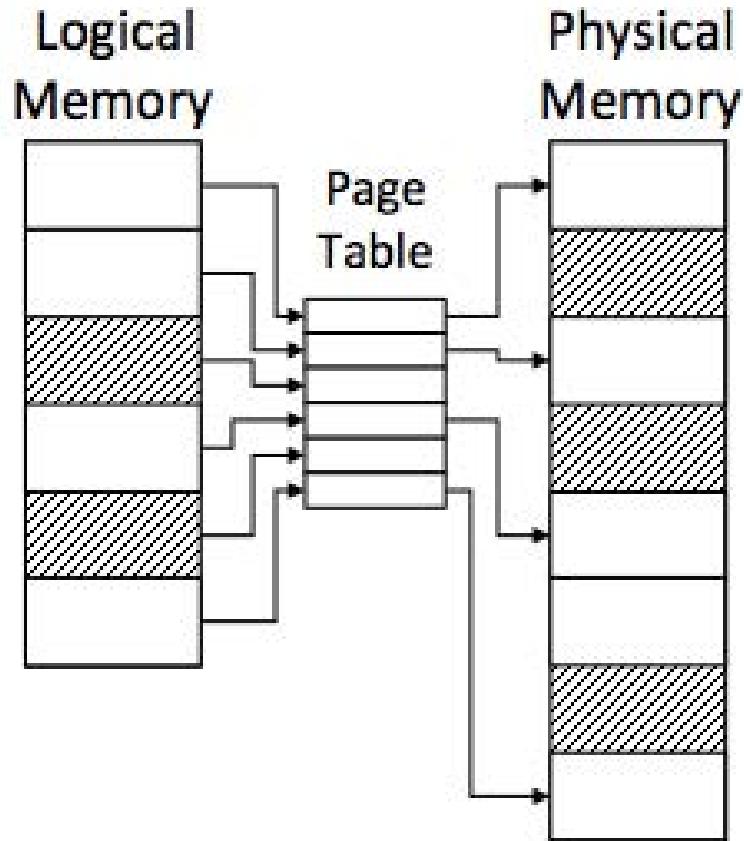


Figure 9.22: The page table maps pages of logical memory to frames of physical memory.

The frame number f is then concatenated to the offset i to produce the physical address $a = f \mid i$ for the logical address (p, i) :

$$a = PT(p) \mid i \quad (9.4)$$

As pages are identified by an integer p , which can range from 0 to $N_L - 1$ (N_L is the maximum number of pages), the $p + 1^{st}$ entry corresponds to page p , e.g., 1st entry corresponds to page 0, the 2nd to page 1, etc. The total number of entries can be up to N_L , but will generally be much less (as N_L is typically a very large number, much larger than what processes will typically need, thus creating the illusion that a process has an unlimited number of pages at its disposal).

A process's page table is located in the kernel (all page tables are managed by the kernel), specifically in the kernel's data area, and thus will be in physical memory, as opposed to being in the form of hardware registers. There are, however, two hardware registers that are required to support a paged memory, and they are the page table base register (PTBR) and the page table size register (PTSR), as shown in Figure 9.23.

The PTBR points to the start of the current process's (the one that is running) page table, and the PTSR indicates how big it is, i.e., how many entries it has. The PTBR register basically tells the hardware where to find address translation information that will be needed for every memory access to allow it to automatically do logical-to-physical address translation.

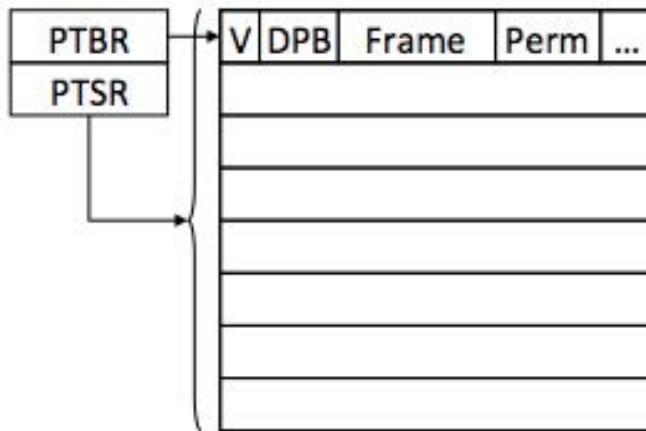


Figure 9.23: A page table, whose location in physical memory is indicated by the PTBR, its size in entries is indicated by the PTSR.

The PTSR register tells the hardware what part of the physical memory corresponds to page table information, thus preventing it from using any memory beyond it.

Each page table entry contains various fields:

- **V**: the valid bit, which if 1, indicates that the entry is valid and thus can be used for address translation, and if 0, indicates that the information is invalid and should not be used
- **DPB**: bits that are used for demand paging (their use will be discussed in the following chapter)
- **Frame**: the frame number to which the page corresponding to this entry maps
- **Perm**: permissions bits, typically three for read, write, and execute

We are presenting what a typical page table looks like, but ultimately, the precise format of a page table is hardware specific. This might include additional information stored in each entry, but that we need not discuss here, as the fields we present are the essential ones to understand how address translation works for a paged logical memory.

When the process is to run, there will be a context switch to it. The kernel will set the PTBR to the start address of the page table, and the PTSR to the number of entries, which corresponds to the number of pages in the logical memory.

Once the process begins running, all memory references, as they appear in the process's memory will be logical addresses in the form (p, i) . The logical address appears as a single word of memory, which is partitioned into an upper set of bits that represent the page number p , and the lower set of bits representing the offset i , as shown in Figure 9.24.

The number of bits for each part (segment and offset) is fixed and determined by the hardware. Since a logical address is stored in a word, say that the word size in bits is w , the page number portion of the logical address is n bits, and the offset portion is k bits, where $n + k = w$. Then the maximum possible number of pages is 2^n , and the page size is 2^k , which is the same for all pages (and frames). For example, for a 32-bit word, partitioned into 20 bits for the page number and 12 bits for the offset, the maximum number of pages is $2^{20} = 1048576$ bytes, and the page size is $2^{12} = 4$ KB.

These are absolute machine limits. The operating system will generally

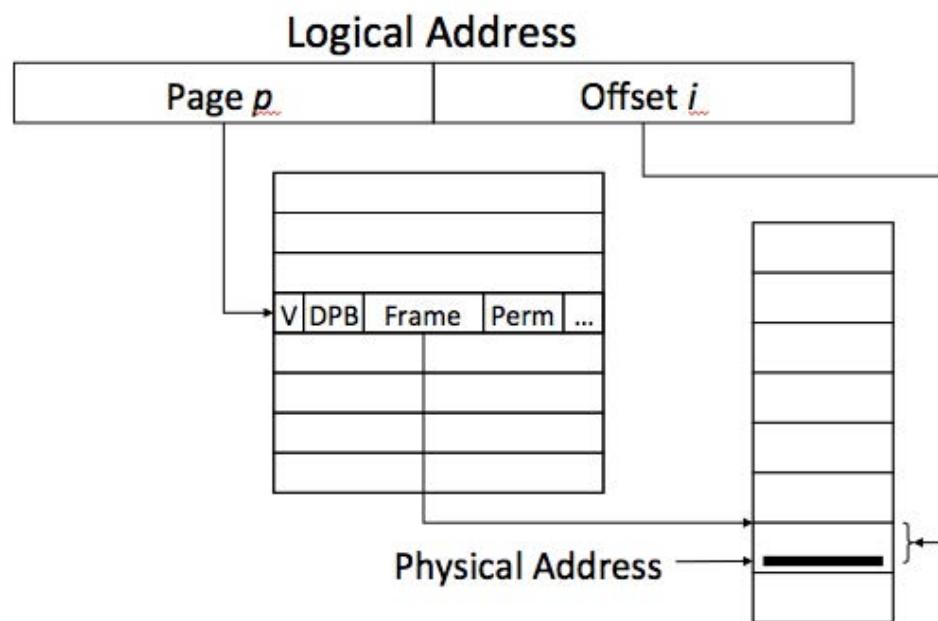


Figure 9.24: A logical address encoded in a word of memory, partitioned into high-order bits representing a page number p and low-order bits representing offset i .

impose its own limit of the maximum to a much smaller number that is reasonable in practice (and which will be the default number of entries in the page table). The reason for this is that the machine design does not presuppose the operating system that will run on it. Some operating systems may make use of many more pages per process than others, and so the machine must provide a maximum adequate for all operating systems expected to run on it.

Say that machine-determined maximum number of pages is 1048576 (i.e., $n = 20$). If an operating system, however, generally uses only 10000 pages per process, it can impose its own limit of, say, 16384 (2^{14}) pages. This means that page tables will be limited to 16384 entries, rather than 1048576 entries if the hardware maximum was used, which is a significant savings. Consequently, only 14 (to encode a number for up to 16384 segments) of the 20 bits in the page number portion of the logical address will be used; the others are essentially wasted.

We now resume our discussion of how address translation occurs. The hardware strips the high-order bits containing the page number from the word containing the logical address, which it will use it to index into the page table (which it finds via the PTBR). It will check to make sure the page number references an entry within the confines of the page table, and so it tests whether the page number is less than contents of the PTSR, which was previously set with the number of possible entries in the page table, as shown in Figure 9.25.

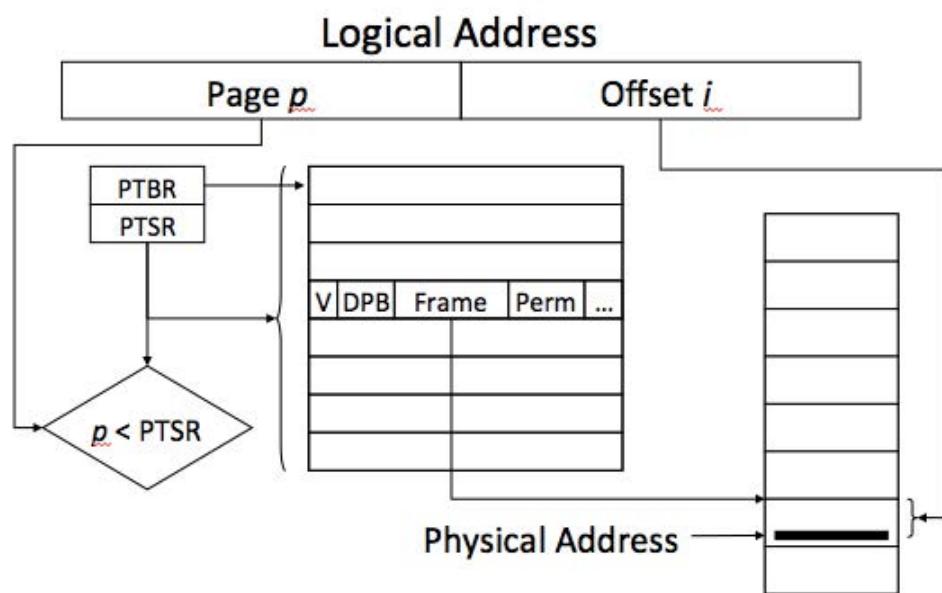


Figure 9.25: The page number p must be less than the contents of the PTSR, which contains the total number of entries in the page table.

If the test fails, the logical address is bad, and cannot be translated, which is an address translation exception. The typical resolution is to trap into the kernel, allow the kernel to terminate the process, or to execute an exception handler that was pre-registered by the process with the kernel.

If the test passes, the hardware uses the page number p to index into the page table. This is done by multiplying p by the number of bytes per entry, the latter of which is usually a power of 2, and if so, the multiplication can be very efficiently achieved by shifting p to the left by that power of 2 rather than actual multiplying. For example, if each entry required 4 bytes, the page number p would be shifted to the left by 2 bits. This is then added to the contents of the PTBR, with the resulting address being the entry for page p .

As a hardware optimization, the test that checks whether p is less than the PTSR and the calculation to determine the entry address are done in parallel, with the latter being discarded if the test fails. Then, as described above, the failed test would cause an exception that is then handled by the kernel, resulting in the termination of the process or the execution of an exception handler.

Now that the page's entry has been determined, the hardware inspects the various fields. The first is the valid bit, which indicates whether the entry has valid information in it, as determined by the kernel. The kernel had previously filled the fields with information, indicating where the page is located in physical memory (frame), and the allowable permissions (perm),

and finally sets the valid bit to indicate it is set to 1. Consequently, the hardware checks the valid bit, as shown in Figure 9.26.

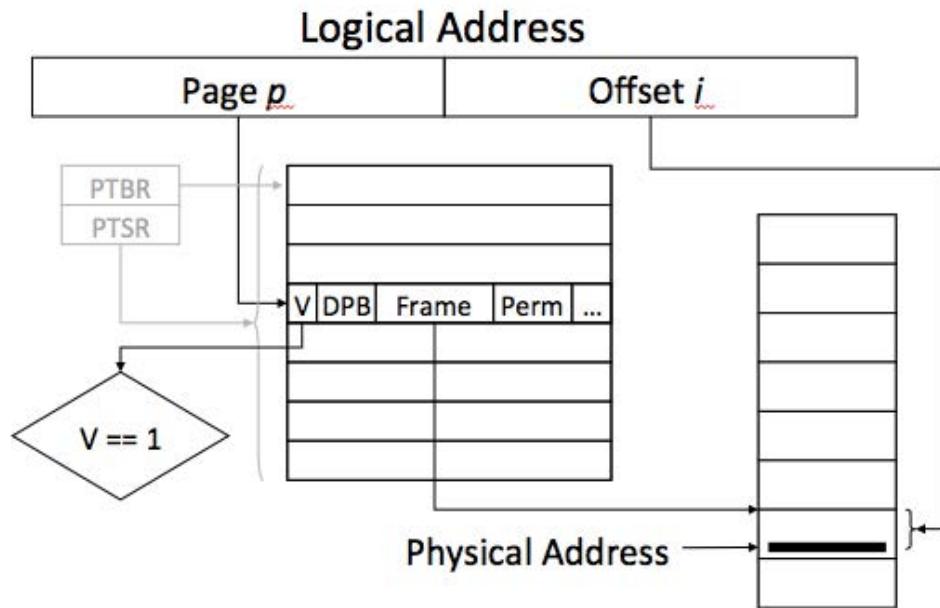


Figure 9.26: Indexing the entry corresponding to the page, and checking the valid bit.

If the valid bit is 0, it means the kernel did not set the entry, and so the information cannot be relied on. If this is the case, control goes to the kernel and the kernel resolves the problem. The question as to why the kernel would *not* have filled the information in the entry will be deferred to the next chapter. Otherwise, if the valid bit is 1, then the entry has good information, and the hardware address translation continues.

If this were segmented memory translation, the offset i would be checked to make sure it is within the segment. But for paged memory translation,

this is unnecessary! The offset i *must* be within the page, as if the number of bits for the offset field is k , then the page size is 2^k , and so that offset cannot possibly refer to any address beyond the page.

Next, the operation that is to be performed on the logical address – reading, writing, or executing, its contents – is considered as to whether it is permitted, by checking whether the corresponding permission bit is set to 1, as shown in Figure 9.27.

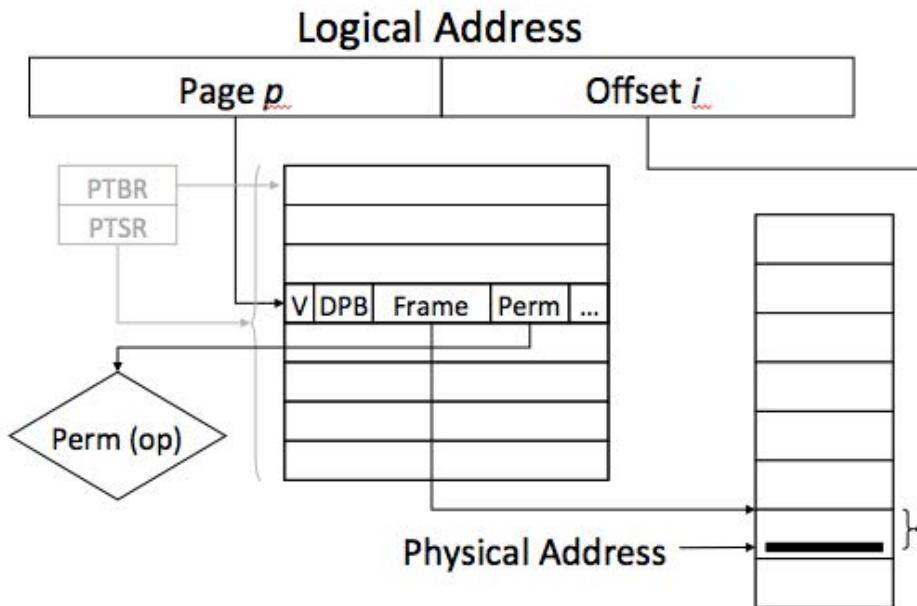


Figure 9.27: Check to see whether the memory operation is permitted.

If the test fails, an exception is generated, indicating that the operation is not permitted (this is a different exception than a bad address), and control goes to the kernel. Such an exception will generally cause the process to be terminated. If the test succeeds, the operation is permitted, and the

hardware address translation continues.

Finally, the value in the frame field, which indicates the frame number to which the page number is mapped, is *concatenated* to the offset. This is what actually converts the logical address into a physical address, as shown in Figure 9.28.

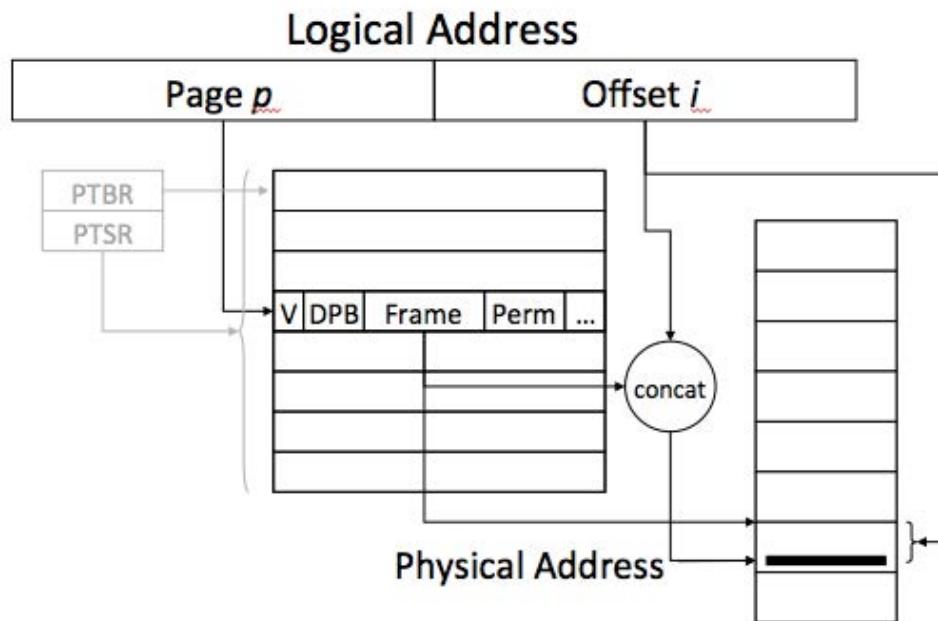


Figure 9.28: The frame number and offset are concatenated to get the physical address.

Another view, emphasizing how concatenating essentially replaces the page number with the frame number in the resulting physical address, is shown in Figure 9.29.

This address can now be submitted to the physical memory, and the operation that generated this memory reference will be applied.

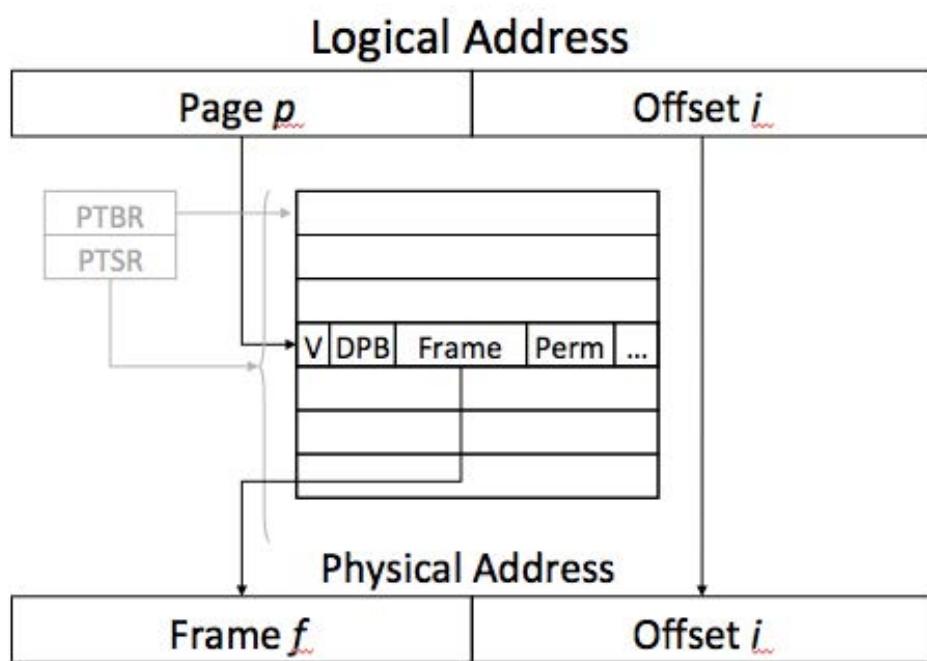


Figure 9.29: The frame number replaces the page number to get the physical address.

All of the steps from the indexing into the page table by adding the value in the PTBR to the appropriately shifted page number, to the various tests, to the determination of the physical address and finally submitting it to the physical memory, will be done by the hardware. As this is a significant amount of work that has to occur for every memory reference, as much as possible is done in parallel; but even so, the overhead of address translation is non-trivial. We will return to this point later.

How big can a process's page table be? We can determine its size by calculating the page table's length and width, and multiplying them. To determine the length and width, we need to be given certain parameters that will be dictated by the hardware: the format of the logical address, specifically the number of bits for the page number and for the offset; and the maximum size of the physical memory. These parameters will give us enough information to determine the page table's size, as shown in Figure 9.30.

Let's go through the details of how the size is determined based on these parameters. Given a logical address where the number of bits for the page number is n , the maximum number of entries is 2^n . This tells us one dimension of the page table, its length. If we can determine the width, we can calculate the size.

To find the width, we need to determine the number of bits for each field. The valid bit contributes 1 bit. The frame number determines in which frame the page is placed in physical memory; consequently, it must have enough bits to address any frame in physical memory, and so depends on the maximum

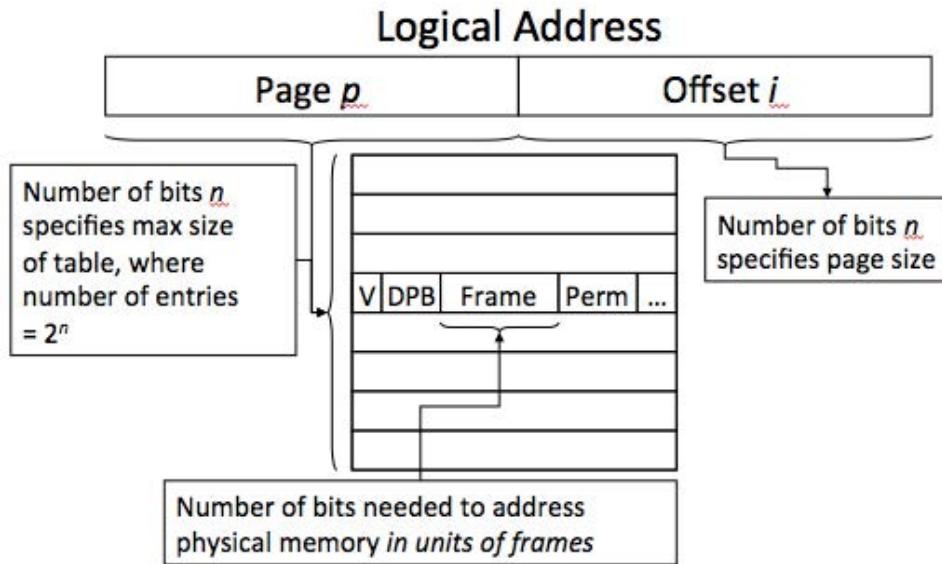


Figure 9.30: Parameters affecting the calculation of the page table size.

number of frames of the physical memory. Assuming a maximum physical memory size of M , if the frame size (which is the same as the page size) is 2^k , then the number of bits needed to address any frame is $\log_2 M/2^k$, which equals the $\log_2 M - k$.

Next we have the permission bits. For this, we will assume 3 bits, to express read, write, and execute permissions.

Finally, we note that a page table entry should be an integral number of bytes, and better yet, a power-of-2 number of bytes, for the same reasons as we gave for a segment table entry, i.e., for fast indexing. For example, if the number of bits were 32, which corresponds to 4 bytes, then if we want to index to page table entry j , the starting byte of this entry is simply j shifted

left by 2, which is a fast way of multiplying j by 4, the size of each entry. Consequently, we will dedicate p bits for padding such that the total size of the entry is a power-of-2 number of bits.

Summarizing what we have so far for the number of bits for a page table entry, i.e., the width of the page table:

- 1 bit for the valid bit
- 2 bits for demand paging
- $\log_2 M - k$ bits for the base field, where M is the maximum size of physical memory and 2^k is the size of a frame
- 3 bits for the permissions field
- p bits for padding to make the width a power-of-2 number of bits

and so, adding all the bits, and dividing by 8 to convert to bytes, the width of the page table is $(1 + 2 + \log_2 M - k + 3 + p)/8$.

Recalling that the maximum length of the page table is 2^n , where n is the number of bits for the page number in the logical address, we can multiply this length by the width we just determined to arrive at a formula for sizing the page table:

$$\text{page table size} = 2^n \times (1 + \log_2 M - k + 3 + p)/8 \quad (9.5)$$

Let's consider an example. Given the following parameters (determined by hardware):

- A logical address of 32 bits, with 20 bits for the page number and 12 bits for the offset
- A maximum physical memory size of 8 GB

From these parameters, we can determine the sizes of the various fields:

- 1 bit for the valid bit
- 2 bits for demand paging
- 21 bits for the frame number field, since $\log_2 8 \text{ GB} - 12 = 30$
- 3 bits for the permissions field
- 5 bits for padding, since the total of the above is 27, and adding 5 bits gets us to 32, a power of 2

Consequently, the width of the page table will be 4 bytes.

The length of the page table is $2^{20} = 1048576$, since 20 bits are used for the page number in the logical address. Thus, the size of the page table in our example is $1048576 \times 4 (\text{L} \times \text{W}) = 4 \text{ MB}$. This calculation is summarized in Figure 9.31.

We conclude our discussion on a paged logical memory by reviewing its advantages and disadvantages. The major advantage is that all pages are the same size, and that a frame is the same size as a page, allowing any page to fit in any frame. So, their allocation requests correspond to the same-size approach for allocation. And as we reviewed in the last chapter,

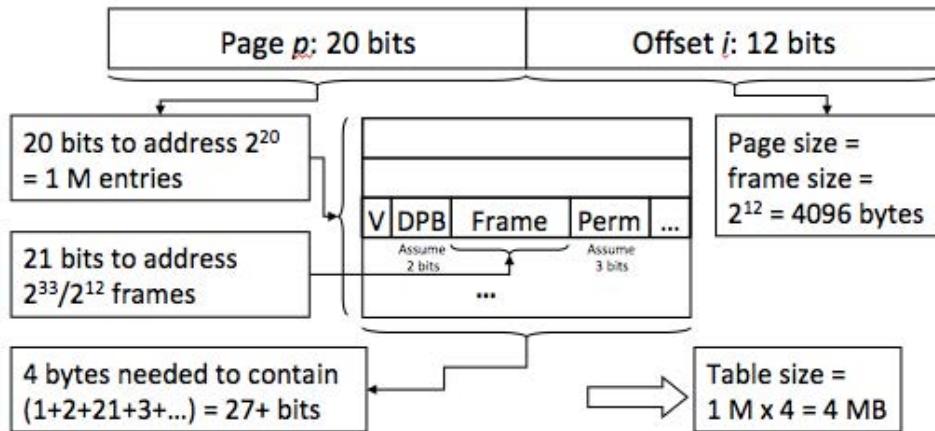


Figure 9.31: An example of sizing the page table.

same-size allocation is very fast, as there is no need to “find a large-enough hole.” All holes, which are frames, are the same size, and any frame will do. Consequently, there is no external fragmentation. However, there is internal fragmentation. Since the boundaries of a page cannot be adjusted, some pages will be only partially used; the remainder of the page is wasted, i.e., it is an internal fragment. And so while a paged logical memory has the advantage of fast and simple allocation, it has the disadvantage of internal fragmentation.

The more significant disadvantage is that pages, while good physical units of memory (because allocation is simple and fast), are not good logical units of memory, as their size cannot be adjusted to the type of information they contain. The boundaries of a page are fixed according to their specified single size, rather than according to the start and end of the same type of

information, such as code, data, or stack. A typical page size, such as 1 KB, makes it highly unlikely that a page will be able to store all the code, or all the data, or the entire stack. In general, many pages will have to be devoted to each of these memory areas. Consequently, sharing a page doesn't make much sense, as that page will only contain a small portion of a memory area; we would have to share many such pages.

However, consider that a page may happen to store portions of multiple areas of memory. For example, it just may happen that a page straddles the ending portion of the code and the starting portion of the data. Sharing now becomes very complex, if not impossible to do in a rational way. Furthermore, permissions also become problematic. If a single page stores some code and some data, we must set its permissions to read, write, and execute, despite that only a portion should execute apply while only the other portion should read and write apply.

Interestingly, the advantages and disadvantages of segments and pages are inverses of each other! Segments are good logical units of information (because they can be sized and permissions can be set according to their contents), whereas pages are not (because they can't be sized appropriately). But pages allow for fast allocation (because any page will fit in any frame; they are all the same size) whereas segments do not (because a large-enough hole must be found given their potentially different sizes). Segments lead to external, but not internal, fragmentation. Pages lead to internal, but not external, fragmentation.

Which is best? From a user’s (programmer’s) point of view, having segments is superior. Being able to create a segment to contain a specific type of information, to size it and set its permissions as needed, and to be able to share it (or not) independent of other segments, is very useful. But from the system’s point of view, pages are superior. Allowing for allocation of memory by being able to assign a page to any free frame is so much simpler, and faster, than dealing with the complexity of different-sized blocks of memory. There is no issue of determining the “best” hole from which to allocate, or dealing with tiny holes that are effectively useless.

Fortunately, we don’t have to choose one or the other. We can choose to have both.

9.8 Segmented Paged Memory

A segmented paged memory is a two-level memory scheme that combines segments and pages: we have segments, with each segment composed of a set of pages. Consequently, from the user’s point of view, the logical memory looks like a set of segments, with all of its advantages. A segment can be sized accordingly, set with appropriate permissions, and sharable as a unit. However, unbeknownst to the user, a segment is composed of a set of pages. Consequently, when physical memory is allocated for a segment, what is actually happening is that a set of frames are allocated for the pages that compose the segment. Thus, we gain the benefit of simple and fast memory

allocation, and avoid the problems of external fragmentation.

To implement a segmented paged memory, we will use the mechanisms we have already discussed for each: segmented address translation using segment tables, and paged address translation using page tables, as shown in Figure 9.32.

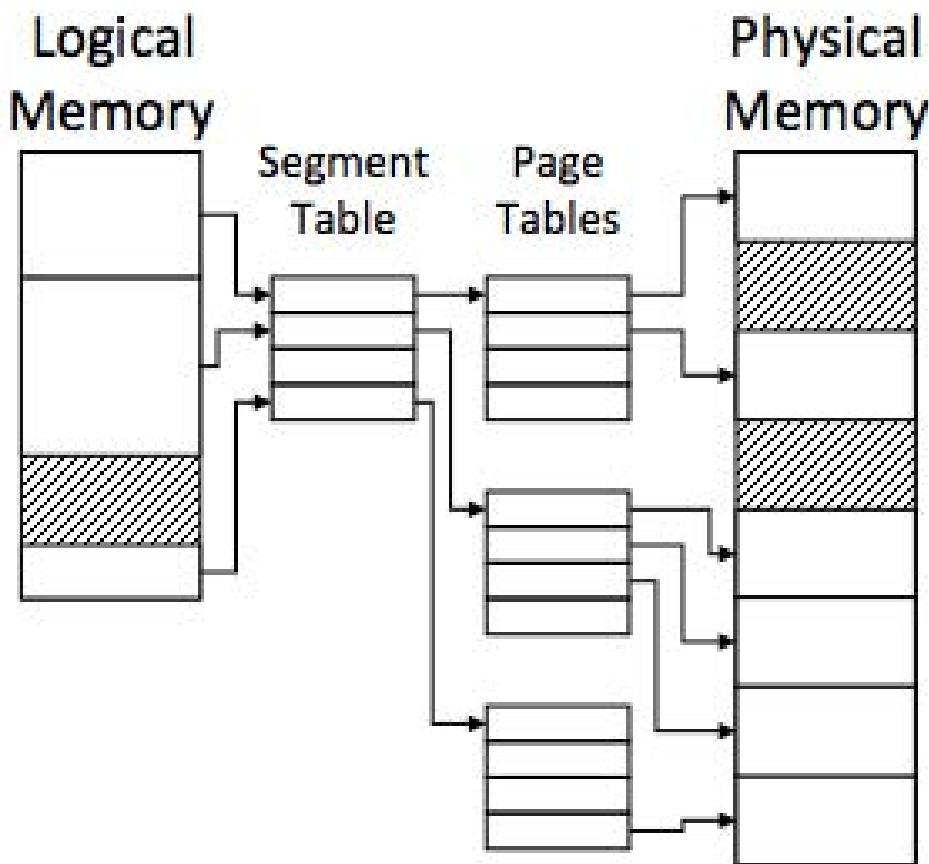


Figure 9.32: A segmented paged logical memory, implemented using a segment table and page tables, one per segment.

The address translation from logical address to physical address will be

a two-level *composition* of segmented translation, and *then* an application of paged address translation. Consequently, the logical address is hierarchical, where the high level view of a logical address is of the form (s, i_s) , where s is the segment number and i_s is the offset within the segment. However, the low level view is that the offset i_s within the segment is actually further broken down into a page number and offset within that page, (p, i) , because a segment is actually composed of pages. This becomes evident in the form of the bit string of the segment offset i_s , which is partitioned into high-order and low-order bits, the former being the page number of the page within the segment, and the latter being the offset within that page. Consequently, the logical address as (s, i_s) expands to $(s, (p, i))$, which we can convey graphically as in Figure 9.33.

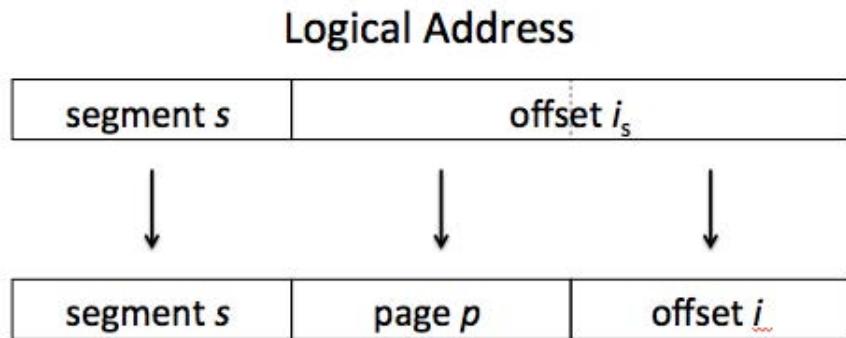


Figure 9.33: A logical address in segmented paged memory is a composition of segments and pages, where the logical address is hierarchical: at a high level, it looks like a segment address (s, i_s) , but at a low level, the offset is further broken down into a page address (p, i) .

The address translation from a segmented paged logical address to a phys-

ical address follows the steps of segmented address translation followed by paged address translation that we already covered, including all the various tests, which we now go through.

The hardware first strips the high-order bits containing the segment number from the word containing the logical address ($s, (p, i)$) shown in Figure 9.33, which it will use it to index into the segment table (which it finds via the STBR). It will check to make sure the segment number references an entry within the confines of the segment table, and so it tests whether the segment number is less than contents of the STSR, which was previously set with the number of possible entries in the segment table, as was shown in Figure 9.13.

If the test fails, the logical address is bad, and cannot be translated; this is an address translation exception. A typical resolution to this problem is to trap into the kernel, which then allows the kernel to terminate the process, or to execute an exception handler that was pre-registered by the process with the kernel.

If the test passes, the hardware uses the segment number s to index into the segment table, which is done by multiplying s by the number of bytes per entry. This is then added to the contents of the STBR, with the resulting address being the entry for segment s .

Now that the segment's entry has been determined, the hardware inspects the various fields. The first is the valid bit, which indicates whether the entry has valid information in it, as determined by the kernel. The hardware checks

the valid bit, as was shown in Figure 9.14.

If the valid bit is 0, it means the kernel did not fill the entry, and so the information cannot be relied on. If this is the case, control goes to the kernel and the kernel will resolve the problem. If the valid bit is 1, then the entry has good information, and the hardware address translation continues.

Next, the contents of the bound field are checked. This is one place where things change a bit from basic segmented address translation that we covered earlier. Recall that in a segmented paged memory, a segment is composed of pages, as shown in Figure 9.32. Consequently, the bound in the segment table entry refers to the maximum number of pages in the page table (that corresponds to this segment), rather than the maximum number of bytes in the segment. The page number is stripped from the middle of the logical address ($s, (p, i)$), and checked to see whether it is less than the contents of the bound field (the maximum number of pages in the page table).

If the test fails, an exception is generated, indicating a bad logical address, control goes to the kernel for resolution. If the test succeeds, the page number is less than the bound value, and the hardware address translation continues.

Next, the operation that is to be performed on the logical address – reading, writing, or executing, its contents – is considered as to whether it is permitted, by checking whether the corresponding permission bit is set to 1, as was shown in Figure 9.16.

If the test fails, an exception is generated, indicating that the operation is not permitted (this is a different exception than a bad address), and con-

trol goes to the kernel for resolution. If the test succeeds, the operation is permitted, and the hardware address translation continues.

Next, the value in the base field is extracted. This value indicates the location of the segment's page table in physical memory. At this point, the segment portion of the translation is complete, and we now proceed to the page portion.

The hardware uses the page number p to index into the page table, thus locating the entry for page p . The hardware can now inspect the various fields in the page table entry. The first is the valid bit, which indicates whether the entry has valid information in it, as determined by the kernel, as was shown in Figure 9.26.

If the valid bit is 0, it means the kernel did not fill the entry, and so the information cannot be relied on. If this is the case, control goes to the kernel and the kernel will resolve the problem. Otherwise, if the valid bit is 1, then the entry has good information, and the hardware address translation continues.

Next, the operation that is to be performed on the logical address – reading, writing, or executing, its contents – is considered as to whether it is permitted, by checking whether the corresponding permission bit is set to 1, as was shown in Figure 9.27. Note that this permission check is done twice, once at the segment level, and the other at the page level. This seems, and generally is, redundant, but does allow for one to be different from the other. This might happen in the case of a shared segment, where for one

process, the segment table entry has one set of permissions, say **rw-** (read and write, but not execute) and the segment table entry for another process has a different set, say **r--** (read only). But, both segment table entries point to the same page table, which can only express one set of permissions, say **rw-**. Consequently, for one process, only reads are permitted, while for the other, reads and write are permitted.

If the permissions test fails, an exception is generated, indicating that the operation is not permitted, and control goes to the kernel for resolution. If the test succeeds, the operation is permitted, and the hardware address translation continues.

Finally, the value in the frame field, which indicates the frame number to which the page number is mapped, is concatenated to the offset i in $(s, (p, i))$. This converts the logical segmented paged address into a physical address, as shown in Figure 9.34. This physical address can now be submitted to the physical memory, and the operation that generated this memory reference will be applied.

With a segmented paged logical memory, we have:

- A view of memory to the user as a collection of segments. Each segment is a separate memory, which can be sized according to the user's needs, and limited in what operations are permitted, e.g., read, write, execute.

Finally, these segments can be shared between processes.

- Allocation is based on pages, which are fixed in size and all the same

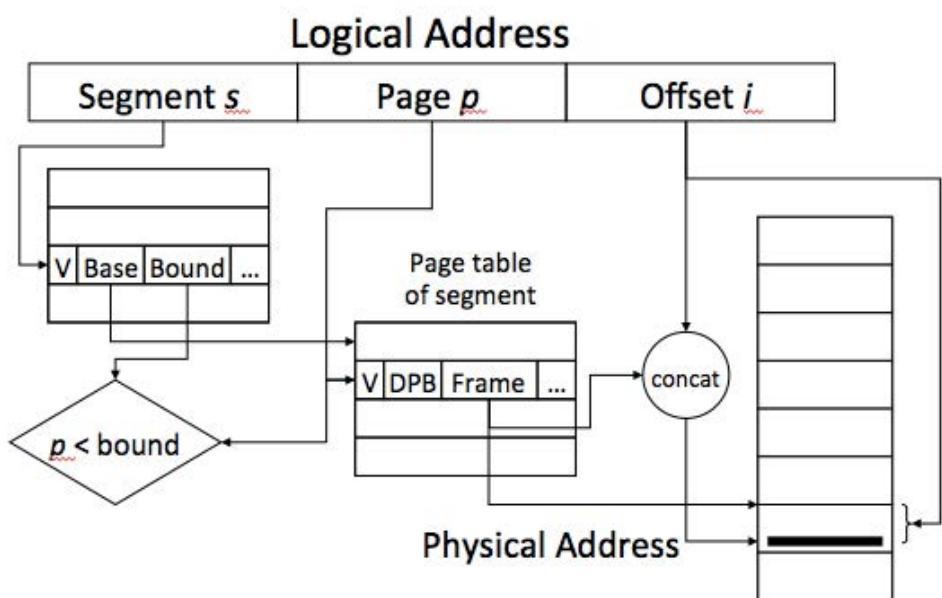


Figure 9.34: Converting a segmented paged logical address to a physical address.

size, and perfectly match (in size) their physical memory counterparts, frames, such that any page will fit exactly in any frame. This makes allocation very simple; if n pages need to be allocated, as long as there are n frames, the allocation will succeed, regardless of where the pages are located in logical memory and regardless of where the frames are located in physical memory.

- Despite the structuring of logical memory as segments (as viewed by the user), there is no external fragmentation, because segments are allocated as a set of pages. Consequently, their sizes are not exactly as requested by the user, but rather the minimum number of integral pages that will satisfy the request. Thus, there will be internal fragmentation, which will be one-half the page size per segment, on average. This is because we cannot assume the last page of a segment will be fully utilized. If we assume an equal likelihood for each different amount of the last page being utilized, then, on average, one half of it will be utilized.
- Given the tradeoff of allocation speed and memory use efficiency, we favor a faster allocation (resulting from allocating same-sized units) than memory use efficiency (wasting one-half a page per segment), based on the view that CPU time is expensive and memory space is cheap.

9.9 Cost of Translation

A segmented paged memory (or even just a segmented or paged memory) seems to have many desirable properties, but they come at a significant cost. Consider that, without a logical memory, a read or write operation to physical memory costs some amount of time – call it T_{mem} . But if we have a logical memory, where a logical address must be translated into a physical address, that translation will take some time, in addition to T_{mem} to actually read or write the physical memory. We are imposing a cost⁴ that is incurred *on every single memory reference*. How much time does address translation add?

Consider the various steps involved in translating a logical to physical address. There is the locating of the entry in the segment or page table. There is the reading of the entry to do the various tests to see whether the address or operation is valid, and then to obtain the base value (for segments) or the frame number (for pages). Since the segment tables and page tables are themselves in physical memory, this reading of the table entry costs at least T_{mem} , the basic time to read or write memory. In other words, for every memory reference done by a program, an additional memory reference is needed to first access the segment table or page table, or in the case of a segmented paged memory, two memory references to access the segment table *and* page table, to do the translation.

⁴Here our main concern is cost in time. But, there are also other costs: the cost of the chip area devoted to implementing address translation, the memory space needed in terms of segment tables and page tables, etc. However, loss of time is so critical, and generally overwhelms the other costs, that it is time that we focus on

Consequently, what was just one T_{mem} becomes two or three T_{mems} per memory access. The speed of running a program is roughly dictated by the rate of memory references it is making (since memory accesses cost much more than CPU logic or simple arithmetic operations), by adding logical memory translation, we are effectively slowing down our machine by one half or one third! No one would buy a computer if they were told that it operates at half or one-third speed, but it's worth it because it comes with logical-to-physical memory translation.

Fortunately, there is a solution to this problem. It is based on taking advantage of a property of memory access behavior of programs called locality of reference. Most programs have this property, as it is derived from good programming practices (i.e., “structured programming”), compiler techniques, and even hardware architecture design. We will simply assume programs have this property, and we will focus on how to take advantage of it.

When a program runs, memory is generally not accessed uniformly. Rather, there will be hot spots, i.e., concentrations of accesses to certain parts of memory. We call this *locality of reference*. Observing the sequence of addresses of the memory accesses and looking for certain patterns of repetition determines the presence of locality of reference. For example, if there is a reference to address x , there is a high likelihood that in the accesses that follow (even the very next one), there will be another reference to the same address x . This is locality of reference in *time*. Also, if there is a reference to address x , there is a high likelihood that in the accesses that follow (even

the very next one), there will be a reference to a nearby address $x + a$, where a is small, e.g., within a page. This is locality of reference in *space*.

If there is locality of reference, in time or space or both, there will be a high concentration of a small set of addresses in a long sequence of accesses. In other words, most accesses will be to a small set of memory addresses. Note that these are *logical* addresses. (Locality of reference also exists in the sequence of physical memory accesses, but our concern here is logical memory.) If only we knew what these common addresses were, we could record them in a fast memory that associates each logical address with the physical address to which it translates. Such a fast memory might exceed the speed of standard physical memory by at least an order of magnitude, though it would cost much more than standard physical memory, and so there would be much less of it. This fast memory is called a cache, and a cache that is used for translating logical to physical addresses is called a *table lookaside buffer*, or *TLB*.

A TLB works as follows. We will assume that the physical memory is organized as a sequence of frames; consequently, the logical address is either of a paged memory or segmented paged memory. Purely segmented logical memories are not common, but the same idea as what we describe would apply.

A TLB is structured as a table of pairs of numbers: a “key” and a frame number. Every time there is a memory reference, which requires a logical-to-physical memory address translation, the hardware does the following. It

extracts the higher-order bits from the logical address, i.e., the page number or the combination of segment and page number, and uses those bits to check for a match amongst all the keys, which will occur in parallel for speed. If there is a match, the frame number is extracted from the table and concatenated with the offset (from the logical address) to form the physical address. This is shown in Figure 9.35. All this is done in a small fraction of time that it takes to access physical memory.

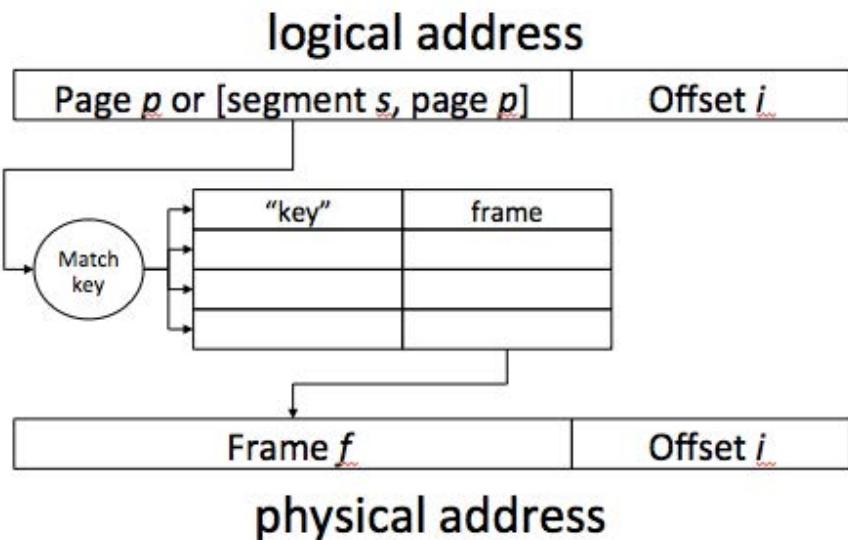


Figure 9.35: The TLB (translation lookaside buffer) is a fast memory cache that contains recent translations of page number for paged logical memory, or segment number and page number for segmented paged logical memory, to physical frame number.

Consequently, if there is a TLB “hit,” the TLB is able to provide the frame number and thus complete the translation of logical to physical address. Let’s call the amount of time this takes as T_{TLB} .

If there is a TLB “miss,” i.e., there is no match of any key in the TLB, then the hardware must do the translation as we discussed earlier, the “long way,” using the regular page table for paged logical memory, or segment table and page table for segmented paged logical memory. The time this will take will be at least the time to access physical memory, T_{mem} , because these tables are being accessed in physical memory. We say “at least” because in addition to accessing the page table entry or segment table entry, there are other operations involved (comparisons, shifting, concatenating, adding, etc.), but as these are small compared to accessing memory, we will use the optimistic value of T_{mem} .

When there is a TLB miss, the TLB will then replace one of its entries with the just-completed translation, thus always holding the most recent translations. This allows the hardware memory address translation system to exploit programs that exhibit a high degree of locality of reference. The entry that is “kicked out” of the TLB will follow a least recently used (LRU) policy: remove the entry that was used furthest in the past, again trying to maximize performance given locality of reference.

To see the gain in performance, let’s consider some actual numbers for T_{mem} and T_{TLB} . We will assume a physical memory access time of $T_{mem} = 100$ nsec, and a TLB access time of $T_{TLB} = 5$ nsec, not unreasonable numbers for the time of this writing in 2020. Let’s further assume a TLB “hit rate” of 99%, i.e., 99 out of 100 times on average, the TLB will be able to successfully translate a logical address to a physical address. Finally, let’s

assume we have a segmented paged logical memory.

If there is TLB hit, the time for translation is $T_{TLB} = 5$ nsec. If there is a TLB miss, then there needs to be a lookup in the segment table, which will take $T_{mem} = 100$ nsec, and then a lookup in the page table, which will take another $T_{mem} = 100$ nsec, totaling 200 nsec.

To each of the above situations, we must add another $T_{mem} = 100$ nsec to do the actual memory operation, i.e., now that we have a physical address, access the contents for the physical memory at that address. Thus, the time to complete a memory operation if there is a TLB hit is $T_{TLB} + T_{mem} = 105$ nsec. The time to complete a memory operation if there is a TLB miss is $3T_{mem} = 300$ nsec. Furthermore, TLB hits happens 99% of the time, and TLB misses happen 1% of the time.

We can now compute an effective memory access time as follows: $99\% \times 105$ nsec $+ 1\% \times 300$ nsec $= 106.95$ nsec. Comparing this to the base physical memory access time of 100 nsec, we see that the cost of logical-to-physical address translation for a segmented paged memory is about a 7% slowdown of our program, compared to running our program on a machine where all the addresses generated by the compiler are physical addresses. A 7% slowdown is a small and acceptable cost when balanced against all the advantages we get from a segmented paged logical memory.

Keep in mind that a TLB will hold recent logical-to-physical memory address translations for the *currently running process*. When there is a context switch, the translations in the TLB are no longer valid for the next process,

and so the TLB must be cleared, or “flushed.” The TLB will then incrementally be filled as the process executes. It is a curious fact that, during the beginning a quantum, a process runs “slow” because there will be many TLB misses, but as the TLB fills up, it will run progressively faster. There are alternative TLB designs, where each entry is tagged with a process ID. Thus, when a process runs, entries marked with another process’s ID will not be used for translation, which means that the TLB need not be flushed. However, when there are TLB misses, entries that “belong” to other processes can be used for the current process, making the TLB more adaptable.

9.10 Summary

In this chapter, we learned that a logical memory is a valuable abstraction in that it allows processes to organize their view of memory as a set of many separate (separately named, separately addressable, etc.) memories, each of which can be used for a different purpose. This logical memory is independent of the physical memory; in particular, the logical address space can be assumed to be independent of the physical address space, allowing a compiler to generate memory references, which are logical, without concern as to how the logical memory is realized, i.e., using the physical memory to store its various parts.

Logical memory can be organized as units that can be separately sized, which are called segments. Each segment has a unique number, can be sized

according to need, can grow and shrink independently of other segments, can be limited to what operations are permitted (e.g., read, write, execute), and can even be shared between processes. However, a segmented logical memory will suffer from external fragmentation. A segmented logical address must be translated to a physical address, which is implemented using a per-process segment table. Logical memory can alternatively be organized as a sequence of units that are all the same size, which are called pages. By also organizing physical memory as a sequence of frames, each of which is the same size as a page, any page will fit in any frame, making allocation simple and fast. A paged logical memory will not suffer from external fragmentation, but will suffer from internal fragmentation. A paged logical address must be translated to a physical address, which is implemented using a per-process page table.

Since each type of organizing logical memory has its advantages, we can combine them to enjoy them all, thus resulting in a segmented paged memory, where each segment is a sequence of pages. Consequently, the user sees a logical memory composed of segments, which can store different types of information, sized separately, and even shared between processes. But allocation will be based on pages, which can be done very efficiently and without resulting in external fragmentation. While there is internal fragmentation, it is limited to half a page per segment, a small amount.

There is a significant cost to translation; every segment or page table lookup requires a physical memory reference that must now be done in ad-

dition to the access of the physical memory required by the actual memory operation. This can result in a slowdown of 200% for segmented memory or a paged memory, or 300% for a segmented paged memory. This problem can be overcome by making use of a TLB (table lookaside buffer) that is a fast memory cache that maintains recent logical-to-physical address translations. A TLB will work well if a program's memory behavior exhibits locality of reference, the property that if there is an access to a certain logical address, there is a high likelihood that the next accesses will be to either that address or ones that are close by. By adding a TLB, the slowdown is reduced from 200% or 300% to less than 10%, and acceptable cost given the advantages of having a logical memory.

9.11 Exercises

1. Regarding the sharing of memory, what is the Addressing Problem? ...
the Protection Problem? ... the Space Problem?
2. What is an address space?
3. What is a physical address space?
4. Where is the kernel typically located, and why is it located there?**
5. What is a logical address space?
6. What is the difference between a logical address space and a physical

- address space?*
7. What does it mean to convert logical addresses to physical addresses – why is this necessary?*
 8. When and how can the conversion be done in software?** ... in hardware?**
 9. What hardware is needed for basic logical-to-physical address translation?*
 10. What does it mean that a base register achieves “relocation”?*
 11. How is a bound register used to achieve protection?* ... protection from what?**
 12. On a context switch, what is saved and restored when using base/bound registers?**
 13. Why must only the kernel load the base and bound registers?*
 14. How is the kernel’s memory protected from processes?***
 15. Why is it worth breaking up a process memory into text, data, and stack when allocating memory?*
 16. What are the two approaches of structuring an address space, and what is their key difference?**
 17. What is a segment?*

18. How is a segment identified?
19. What is the form of a logical address for a segmented memory?
20. In Figure 9.9, how do the segments differ?**
21. What is a segment table?
22. How many segment tables are there?**
23. Where are segment tables located?***
24. How is a logical address converted to a physical address using a segment table?*
25. What are the contents of a segment table entry, and what do each mean?*
26. What is the Segment Table Base Register?
27. What is the Segment Table Size Register?
28. Why must these be in hardware?**
29. What are the steps of converting a logical address to physical address, including the various checks?**
30. What determines how many entries are in a segment table?*
31. What determines how big the base field is in a segment table entry?*
... the bounds field?*

32. What part of a logical address determines the maximum size of a segment, and why?**
33. In Figure 9.19, can you explain how the size of the segment table was determined?*
34. What are the pros and cons of segmentation, and explain the reason for each one?*
35. Is external fragmentation an issue in segmentation, and why or why not?*
36. Is internal fragmentation an issue in segmentation, and why or why not?**
37. What is a page? ... a frame?
38. How is a page identified? ... frame identified?
39. What is the form of a logical address for paged memory?
40. How is the size of a paged logical address space determined?*
41. In Figure 9.20, what is meant by the pointers from logical to physical memory?**
42. What is a page table?
43. How many page tables are there?**

44. Where are page tables located?***
45. How is a logical address converted to a physical address using a page table?*
46. What are the contents of a page table entry, and what do each mean?*
47. In Figure 9.22, what does it mean that a page table maps logical pages to physical frames?
48. What is the Page Table Base Register?
49. What is the Page Table Size Register?
50. Why must these be in hardware?**
51. What are the steps of converting a logical address to physical address, including the various checks?**
52. What determines how many entries are in a page table?*
53. What determines how big the frame field is in a page table entry?*
54. Is the frame field in page table entry more similar to the base field or bounds field in a segment table entry, and why?**
55. What part of a logical address determines the size of a page, and why?**
56. In Figure 9.31, can you explain how the size of the page table was determined?*

57. Why, in general, is a page table so much larger than a segment table?***
58. Is external fragmentation an issue in a paged memory, and why or why not?*
59. Is internal fragmentation an issue in a paged memory, and why or why not?**
60. What is meant by a segment being a good “logical” unit of information?*
61. What is meant by a page being a good “physical” unit of information?*
62. What is the purpose of combining a segmented memory with a paged memory?**
63. Why is such a memory structured as segments of pages, rather than pages of segments?***
64. What are the steps of converting a logical address to physical address when using combined segments and pages, including the various checks?**
65. What comprises the cost of logical-to-physical address translation?*
66. What is meant by “locality of reference”?**
67. What is the purpose of a TLB, and how does it work?**

68. What happens to a logical-to-physical address translation if there is a TLB hit?*
69. What happens if there is a TLB miss?*
70. What determines translation cost when we include a TLB?**
71. How does the size of a TLB affect hit rate, response, and expense, and why?**
72. What is meant by the TLB having to be “flushed” on context switches, why is this necessary, and what problem does this cause?***
73. How does tagging entries with PIDs help, and what is the downside?***



Chapter 10

Virtual Memory

In the last chapter, we developed the abstraction of logical memory. For example, in a segmented paged memory, the logical memory is structured as segments, which themselves are composed of pages. The partitioning into segments promotes convenience in usage, and the partitioning of segments into pages promotes ease of allocation. Relocation is achieved via logical-to-physical address translation, and protection is achieved by checking whether addresses are within bounds and whether operations are permissible.

10.1 Not Everything Needs to be in Memory

In addition to these properties, there is another important implication that derives from having broken up a monolithic logical memory into smaller pieces such as segments and pages, and that is that *not all the pieces need*

to be in physical memory at the same time. A piece of logical memory need only be in physical memory if it is being referenced, otherwise, it can reside on secondary storage, such as a disk. This idea can be exploited by keeping all the pieces on secondary storage, and only bringing in to physical memory those pieces that are needed, i.e., will be referenced.

This is a very important observation because it removes the physical memory as being a limiting factor as to how large a logical memory can be. Consider that a logical memory can be very large, much larger than the physical memory, and as long as those pieces, perhaps even a single piece, that are needed for immediate reference are placed in physical memory, the process can still run. One can imagine arranging for pieces to be moved in and out of physical memory, making sure that only needed pieces are in physical memory. If this could be done, then a process with a very large logical memory, even one that is much larger than the physical memory, can run to completion perfectly well. This idea, called virtual memory, is the subject of this chapter.

What's needed to support this idea? First, we need a way to identify whether a piece of logical memory (segment and/or page) is in physical memory or not. Second, we need a way to move pieces between physical "primary" memory and some external "secondary" storage. Finally, we need the ability to relocate the addresses of the logical memory, which we already have via logical-to-physical address translation.

10.2 From a Logical Memory to a Virtual Memory

Consequently, our *logical memory* becomes a *virtual memory*. A virtual memory is still a logical memory in that it is an abstract memory with a separate logical organization that is separate from physical memory. But, we call it “virtual” because, despite that the logical memory pieces may or may not be in physical memory, they seem to be because when they are needed, we arrange them to be brought into physical memory (and possibly move other pieces out to make room).

Virtual memory is an abstraction that gives the illusion of a potentially very large memory, not limited by the physical memory size. This is achieved by keeping only a portion of the logical memory in physical memory. The rest is kept on disk, which is much larger, and much cheaper, but much slower, than the physical memory. The “pieces” or units of logical memory that are kept in physical memory and moved between the physical memory and disk are segments, or pages, or both.

Let’s first consider a paged virtual memory (later, we will consider a segmented paged virtual memory). A paged virtual memory is similar to a paged logical memory, except that now, not all the pages need be in physical memory. In fact, for a paged virtual memory, all the pages will reside on disk, while some will also reside in physical memory, as shown in Figure 10.1. Determining which should reside in physical memory is a critical decision,

and this will be an important topic in this chapter.

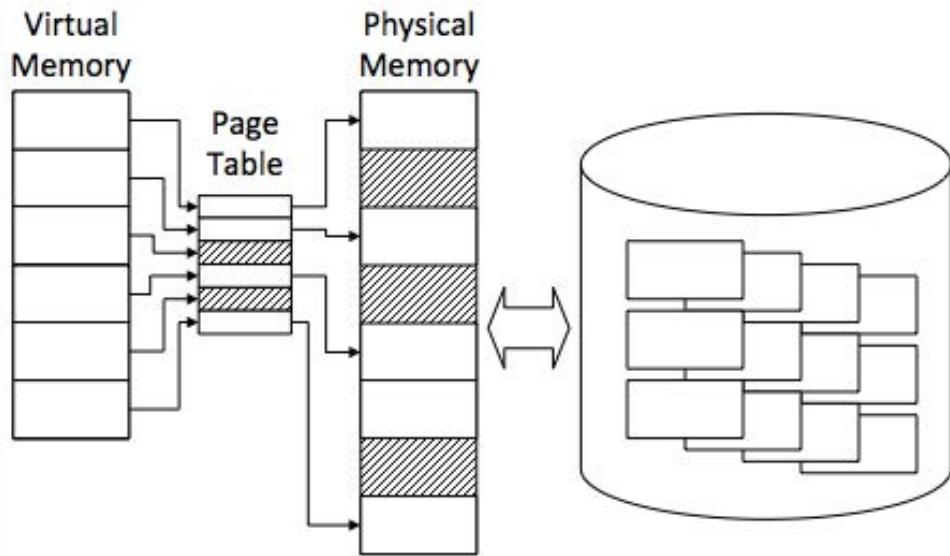


Figure 10.1: A paged virtual memory.

Just like the address space of a logical memory is called a *logical address space*, the address space of a virtual memory is called a *virtual address space*. Translation of virtual addresses to physical addresses is almost the same as logical-to-physical address translation. It has all the steps as logical-to-physical address translation, with one important addition.

10.3 Page Faults

In the last chapter, we saw how to translate a paged logical address, where the hardware uses the page number to index into the process's page table,

to access the page table entry that corresponds to the page being referenced.

A page table entry is shown in Figure 10.2.

Valid	Ref	Mod	Frame number	Prot: rwx

Figure 10.2: A page table entry, now showing the reference (Ref) and modified (Mod) bit, which are demand paging bits (DPB).

Let's review the various fields:

- **Valid:** the valid bit, which if 1, indicates that the entry is valid and thus can be used for address translation, and if 0, indicates that the information is invalid and should not be used
- **Ref:** one of the two demand paging bits (referred to as DPB in the last chapter) that indicates whether the page has been referenced
- **Mod:** the other of the two demand paging bits that indicates whether the page has been modified
- **Frame:** the frame number to which the page corresponding to this entry maps
- **Perm:** permissions bits, typically three for read, write, and execute

To determine the physical address corresponding to the virtual address, the page number must be converted into a frame number, i.e., that of the frame in physical memory that contains the page, which involves accessing the frame field of the page table entry. But before this can be done, there needs to be a check to make sure the entry is valid.

In the last chapter, we assumed the entry would be valid so that the rest of the translation operation could be done. However, it *is* possible that a problem occurs due to a page table entry being invalid, determined by the entry's valid bit being 0. Why would the valid bit be 0, i.e., why would the kernel not have filled the entry with the proper translation information prior to having placed the process in physical memory and then allowing it to run? We asked this question in the last chapter and said we'd defer to this chapter; we are now ready to answer it.

The answer is that the page of memory in question is simply not in physical memory! However, this is not due to a problem with the user program code; there is nothing wrong with the logical, now virtual, address, generated by the compiler. The problem is simply that there is nothing to which the virtual address translates to, *at the moment*.

This event – the virtual address pointing to an entry in the page table whose valid bit is 0 – is called a *page fault*, and will be resolved by the kernel. Upon this occurring, control goes to the kernel. Rather than terminating the process, as might happen with exceptions, the kernel is able to resolve this problem successfully by retrieving the missing page from the disk, bringing it

into physical memory (by finding a free frame, which may involve kicking a page out if there are no free frames; more on this later), recording the frame number in the page table entry (as well as other bits), setting the valid bit to 1, and allowing the process for which the page fault happened to proceed. When that process runs, it will retry the instruction that caused the page fault, but this time, since the valid bit is 1, the translation will succeed and the physical memory can be accessed to allow the instruction to be executed.

10.4 Faults in a Segmented Paged Memory

With a segmented paged virtual memory, the same considerations for invalid table entries apply, for *both* segment tables and page tables. The structure of a segmented paged virtual memory is similar to that of a segmented paged logical memory. Just as in a logical memory, in a segmented paged virtual memory, a segment is composed of a set of pages. But in contrast to a segmented paged logical memory, there are now the possibilities that:

- a page is not in physical memory, causing a page fault
- a segment is not in physical memory, causing a segment fault

Let's go through the translation procedure to see where these faults occur, and how they are resolved. For completeness, we will also include all the various tests to check for address and operation validity. The relationships between segmented paged virtual memory and physical memory, with

the segment and pages tables required for address translation, are shown in Figure 10.3.

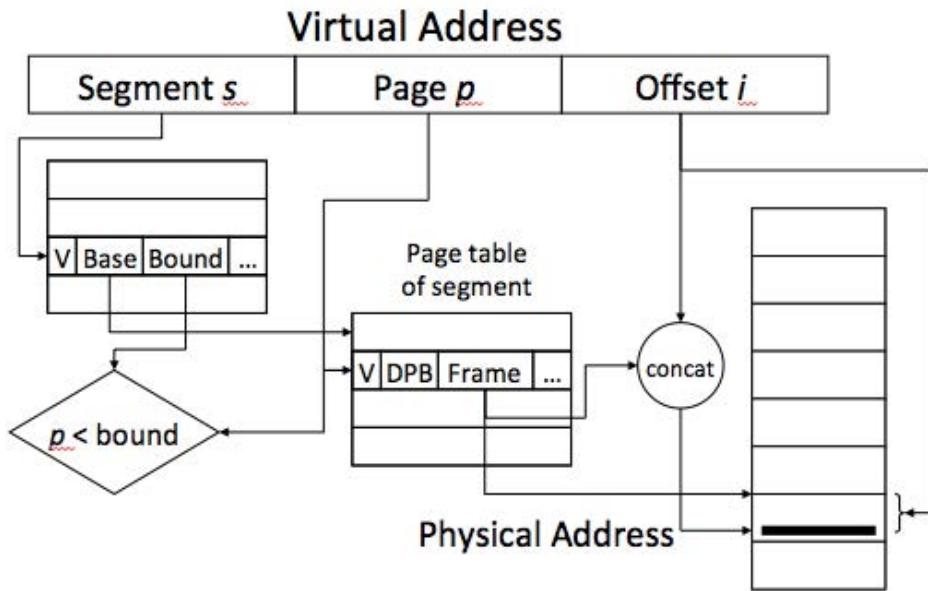


Figure 10.3: Address translation for a segmented paged virtual memory.

The address translation is similar as we described for a segmented paged logical memory. The hardware uses the segment number *s* of the virtual address to index into the segment table, accessing the segment table entry. The location of the segment table in physical memory is known to the hardware because its start address is contained in the STBR (segment table base register, see Chapter 9 for detail of this register as well as the STSR). It will check to make sure the segment number *s* references an entry within the confines of the segment table, which is done by seeing if *s* is less than contents of the STSR (segment table size register).

If the test fails, the virtual address is bad, and cannot be translated, resulting in an address translation exception. This is resolved by automatically (as a result of the exception) trapping into the kernel, which then allows the kernel to terminate the process, or to execute an exception handler that was pre-registered by the process with the kernel. Otherwise, the segment table can be indexed using the segment number s , which allows access of the segment table entry for s .

The valid bit is then checked, which indicates whether the entry has valid information in it, as determined by the kernel. If the valid bit is 0, it means the kernel did not fill the entry, and so the information cannot be relied on. This is a *segment fault*. If this is the case, control goes to the kernel and the kernel will resolve the problem.

A segment fault means that the segment to be accessed – the entire segment, and thus all of its pages – is not in physical memory. In fact, it is the segment's page table that is not present in physical memory, as well as the actual pages in the segment. To handle the segment fault, the kernel will either transfer a page table that was previously created from the disk to the physical memory, or simply create a new page table, store the location of the page table into the segment table entry, and set its valid bit to 1. The kernel then allows the process for which the segment fault happened to proceed. When that process runs, it will retry the instruction that caused the segment fault, but this time, since the valid bit is 1, the translation will be able to continue.

The page number p is then checked to see if it is within the bound for this segment. If not, this is an illegal address, causing an exception, which is handled by the kernel. The kernel will likely terminate the process. Notice the difference between a fault (segment or page) and an exception. The former is not an error of the program and is something the kernel can resolve so that the process can eventually resume, whereas the latter is an error of the program, and the usual resolution by the kernel is to terminate the process. If the page number is within the bound, then the translation can proceed.

Next, the operation that is to be performed on the virtual address – reading, writing, or executing, its contents – is considered as to whether it is permitted, by checking whether the corresponding permission bit is set to 1. If the test fails, an exception is generated, indicating that the operation is not permitted (a different type of exception than an illegal address, but an exception nonetheless), and control goes to the kernel. Such an exception will generally cause the process to be terminated. If the test succeeds, the operation is permitted, and the hardware address translation continues.

The value in the base field of the segment table entry is extracted. This value indicates the location of the segment's page table in physical memory. At this point, the segment portion of the translation is complete, and we now proceed to the page portion of the address translation.

The page number p is now used to index into the page table, thus locating the entry for page p . The valid bit, which indicates whether the page table

entry has valid information in it, is checked. If the valid bit is 0, it means the kernel did not fill the entry, and so the information cannot be relied on. This indicates the occurrence of a page fault. The kernel handles the page fault by retrieving the missing page from the disk, bringing it into physical memory (by finding a free frame, which may involve kicking a page out if there are no free frames), recording the frame number in the page table entry (as well as other bits), setting the valid bit to 1, and allowing the process for which the page fault happened to proceed. When that process runs, it will retry the instruction that caused the page fault, but this time, since the valid bit is 1, the translation will succeed and the physical memory can be accessed to allow the instruction to be executed.

If the valid bit is 1, then the operation that is to be performed on the virtual address – reading, writing, or executing, its contents – is considered as to whether it is permitted, by checking whether the corresponding permission bit is set to 1. If the test fails, an exception is generated, indicating that the operation is not permitted, and control goes to the kernel, which will then generally cause the process to be terminated. If the test succeeds, the operation is permitted, and the hardware address translation continues.

The frame number can be extracted from the page table entry. It is concatenated to the offset, producing a physical address that completes the address translation. Note that the address translation for a segmented paged virtual memory is exactly the same as for a segmented paged logical memory, except that there are now the possibilities of a segment fault and a page fault.

10.5 The Costs of Virtual Memory

Virtual memory sounds too good to be true. We get all the benefits of organization, protection, efficiency of allocation, of logical memory, and, we are not limited by the size of the physical memory. This is achieved by using address translation hardware, and the ability to transfer memory units between the physical memory and a disk, the latter of which can be very large. Is there a catch?

Yes. The catch is that while disks offer lots of memory for very low cost (by a factor of about 10000, or four orders of magnitude, compared to DRAM!), a disk access comes with an immensely large time penalty. While DRAM access time is about 100 nsec, a disk access requires 10 msec, an increase by a factor of 100000, or five orders of magnitude. Every time a page fault occurs, a process incurs this major delay. To put things in perspective, if it took one second to execute an instruction involving accessing memory, then a page fault would delay the process by more than a day! Suffering such delays would be intolerable, unless they happen very rarely. While 10 msec for a disk access is a very long time relative to 100 nsec to access physical memory, it is not a long time in human terms (a 10 msec delay is typically not noticeable to humans, though a delay of a multiple of 10 msec, and certainly when reaching 100 msec, is noticeable). So, as long as accessing the disk occurs very infrequently, we will not notice them.

10.6 Locality of Reference

But how can we control how often the disk is accessed to service page faults?

We can take advantage of the following phenomenon. When a program runs, the various parts of the memory will generally not be referenced uniformly over time. Rather, there will be “hot spots,” i.e., there will be some parts of memory that are accessed much more often than others. If only we knew which parts are the frequently accessed ones, we can arrange to keep those in physical memory.

In fact, memory references will generally cluster in time and in space. A reference to a certain address will either be repeated (time) or will be close by (space). We call this pattern of memory reference behavior *locality of reference*, which we encountered in Chapter 9. Specifically, locality of reference is defined as follows:

- If there is a reference to address x , there is a high likelihood that in the accesses that follow (even the very next one), there will be another reference to the *same* address x . This is locality of reference in **time**.
- If there is a reference to address x , there is a high likelihood that in the accesses that follow (even the very next one), there will be a reference to a *nearby address* $x + a$, where a is small, e.g., within a page. This is locality of reference in **space**.

Most programs exhibit locality of reference when they run. This is due to the way programs are structured. In a well-structured program, blocks of

code localize where memory is accessed. A while loop will contain a block of code that is repeatedly accessed. A data structure packages related variables that are often accessed together. When a procedure is called, the code in its body becomes a “hot spot.” Just the fact that most instructions are executed sequentially means that memory references to access those instructions will be sequential, i.e., located close to each other.

If a program exhibits locality of reference, we can use knowledge of previous accesses to predict where future accesses will be. It won’t be perfect, but for the most part, it will work well. It is only because of locality of reference that virtual memory can be made to work efficiently. Let’s see how.

10.7 Page Replacement

When a process starts running, physical memory must be allocated for it, to store its code, variables, and a stack for activation records, to run. This is true whether we have virtual memory, logical memory, or simply physical memory (i.e., no logical memory or virtual memory). The cost of allocating memory and loading it, to get the process running, will be the same for each, as all will incur the same cost of accessing the disk.

Where there is a significant divergence is for virtual memory, when the physical memory becomes full and a page fault occurs. To bring in the new page from disk, some page in physical memory must be kicked out to make room. This turns out to be a critical decision.

Which page should be removed? Imagine removing a page that, while at some point in the past it was “paged in” because an address within it was referenced, triggering a page fault that caused the kernel to bring it in from disk, will never be accessed again. That is an ideal page to remove, as it is not being used, nor will it ever be used, and yet it is taking up space in physical memory. At the other extreme, imagine removing a page that contains an address that will be referenced in the very near future. When it is accessed, it will cause another page fault. Remember, the goal is to minimize the number of page faults because of the significant cost in delay they incur. Consequently, selecting pages more like the former (that will never be referenced, or if they are to be referenced, will be referenced far in the future), than the latter (that will be referenced in the near future) is what we want.

A *page replacement policy* is a strategy for

- which page(s) to remove
- when to remove it (them)

We want to do this in the cheapest way possible, with the least amount of addition hardware and the least amount of software overhead.

To understand how page replacement policies work, let’s consider three basic ones:

- **FIFO** (first-in-first-out): select the page that is the oldest

- **OPT** (optimal): select the page that will be used furthest in the future
- **LRU** (least recently used): select the page that was used furthest in the past

To see in detail how each of these policies work, we will apply each of them to a *page reference string*. A page reference string is a sequence of page references, which would normally be derived from a memory reference string. Imagine a program running, and while running, recording each and every memory address that is referenced. The sequence of the memory addresses referenced is a memory reference string. Note that these memory references are to virtual memory. For our example, we will assume a paged virtual memory, and so each virtual memory address is composed of a page number and offset. If we strip off the page numbers from each virtual memory reference, the string becomes a page reference string. When evaluating page replacement policies, all we care about is which page is being referenced; the offsets of the memory reference string are irrelevant, and so we can ignore them.

The page reference string we use is shown in Figure 10.4. It is short and unrealistic, and chosen simply for illustrative purposes. However, it allows us to see, for each policy, how and when page faults occur, and over the course of the entire page reference string, how many total page faults occur. Despite its simplicity, the page reference string was chosen such that each policy will exhibit its characteristic behavior. Furthermore, we will assume a physical

memory of only three frames; very unrealistic, but again, kept simple to help explain how the policies work.

Reference string	2	3	2	1	5	2	4	5	3	2	5	2
------------------	---	---	---	---	---	---	---	---	---	---	---	---

Figure 10.4: A reference string of page numbers. The sequence 2, 3, 2, 1, 5, etc. means that in the first time unit, page 2 is referenced; in the second time unit, page 3 is referenced; in the third time unit, page 2 is again referenced; in the fourth, page 1; in the fifth, page 5; etc.

10.8 First In First Out Page Replacement

The *FIFO* policy removes the page that was paged in furthest in the past, and so it is the “oldest” page in physical memory. Its behavior on the page reference string is shown in Figure 10.5.

In this first example, we see that three page faults occur in the first four time units. Since the physical memory begins empty, a reference to a new page will cause a page fault, and that page will simply be placed in the next available frame. We certainly would not want to kick out an existing page if there is a free frame. Thus, page *replacement* only comes into effect when the entire physical memory is full, and thus a page has to be kicked out to make room for a new one.

When page 5 is referenced in the 5th time unit, the physical memory is full, and so the page replacement policy is invoked to determine which page

Reference string	2	3	2	1	5	2	4	5	3	2	5	2
FIFO	2*	2	2	2	5*	5	5	5	3*	3	3	3
	3*	3	3	3	2*	2	2	2	2	5*	5	
9 faults				1*	1	1	4*	4	4	4	4	2*

Figure 10.5: The FIFO page replacement policy. Each vertical column in the lower table represents the contents of physical memory, which has only three frames, at a particular point in time. For instance, in the first time unit, page 2 is referenced (as indicated by the reference string). The physical memory starts with each of the three frames being empty. After the first time unit, the first frame gets filled with page 2, which was due to a page fault, indicated by the asterisk next to the page number 2.

to remove, to make room for page 5. Since page 2, in the 1st frame, is the oldest (the earliest one to be brought in), it is the one that is removed, and page 5 overwrites it.

In the very next time unit, page 2 is referenced. Unfortunately, page 2 was removed in the previous time unit. This is a good example of the oldest page, which was page 2, was not the best page to replace; in fact, in this case, it was the worst page to replace, because it is now being referenced. Consequently, a page has to be removed, and since page 3, in the 2nd frame, is now the oldest page, it is removed, and page 2 overwrites it.

By the end, a total of nine page faults occur. Note that five of them are *obligatory*. When a page is referenced for the first time, it will cause a page fault, no matter what policy is in effect. Given that there are five unique pages in the reference string (page numbers 1, 2, 3, 4, and 5), the

first time each is referenced, they cannot possibly be in the physical memory already, and so we expect a page fault for each one. Note that this does not mean that the choice of policy has no effect prior to these first five faults. In fact, after the third page fault, the policy does make a difference *as to which page is replaced*. This occurs after the third page fault because there are three frames of memory, and so for those first three, the policy makes no difference. In summary, the policy becomes relevant after the number of page faults that equals the size of the physical memory, but the number of page faults for which the policy is responsible is the number beyond the number of unique page references. This is the number we are most concerned about.

In our example, it is the last four of the nine page faults that can be directly attributable to FIFO. Now, is four good, or bad? We really can't say, unless we have a benchmark as to what is the best possible performance that leads to the minimum number of page faults. This is what the next policy, OPT, will tell us. But before proceeding to the next policy, let's review some of the characteristics of FIFO.

One thing that is especially good about FIFO is that it is very simple to implement. The kernel can simply keep a pointer to the next frame to be filled, starting with the first frame. When a page is brought in, it is placed in the frame being pointed to, and the pointer is incremented to point to the next frame. When the maximum numbered frame is filled, the pointer is set to the first frame.

Thus, all that is required to implement FIFO is a pointer and some very

simple code, which is easy to understand. Simplicity is always good for operating system code, as an operating system is software that will live for many years, and will be worked on by many people, most of whom were not involved in the original design. Being able to look at part of an operating system and understand how it works, and how it interacts with other parts, is very important. It is especially important when implementing any policy, because one must be able to reason why the chosen policy is best, or whether a new policy is worth considering. Policies are those parts of the operating system that are replaceable, and may require change if the goal of the system is changed.

While FIFO is simple, and this is a worthy characteristic, unfortunately, it generally does not perform very well. After all, why should the *oldest* page be a good choice for removal? Perhaps the oldest page is also a frequently accessed page; if so, why would we want to remove it? This is precisely what we observed in our example, when page 2 was removed because it was the oldest, not realizing that it would be referenced in the very next time unit.

On the other hand, FIFO may be better than doing nothing. The intuition is that there may be some correlation between old and no longer needed (i.e., will no longer be accessed), and so between this, and the fact that FIFO is so simple to implement, we may do so in the absence of being able to implement a more complex strategy.

10.9 Optimal Page Replacement

The *OPT* policy selects the page that will be used furthest in the future, which may be infinity if it will never be used. This is optimal, the absolute best we can possibly do. After all, if another page will be used sooner, it will cause a page fault sooner than the optimally chosen page, and we want to put off page faults as much as possible. Its behavior on the page reference string is shown in Figure 10.6.

Reference string	2	3	2	1	5	2	4	5	3	2	5	2
OPT	2*	2	2	2	2	2	4*	4	4	2*	2	2
	3*	3	3	3	3	3	3	3	3	3	3	3
6 faults				1*	5*	5	5	5	5	5	5	5

Figure 10.6: The OPT page replacement policy. After the first three obligatory page faults, page 5 is referenced. The page it will replace is page 1, because page 1 is referenced furthest in the future (in fact, never, or at time infinity).

The first four references are to pages 2, 3, 2 again, and 1, and these will simply fill the first three frames. These page faults do not cause any page replacement, so up to this point, there is no difference between OPT and FIFO.

The very next reference is to page 5. This is an obligatory page fault (because page 5 has never been referenced), but the policy now makes a difference, because a page will have to be replaced. OPT chooses to replace

page 1 in the 3rd frame. To see why OPT makes this selection, consider all the possible choices OPT had. It could have chosen pages 1, 2, or 3 (those are the only pages in physical memory). Consider each one, and find when in the future they will be referenced.

Page 2 will be referenced in the very next time unit (after the reference to page 5), and so it is certainly not a good choice. Page 3 is referenced a few time units after page 2. Page 1 is never referenced again. Consequently, it is the best choice! Why select a page to remove that will actually be referenced (and will have to be paged in again), when there is a page that will never be referenced? Consequently, page 5 overwrites page 1, leaving pages 2 and 3 in physical memory.

In the next time unit, page 2 is referenced, and since it is in physical memory, there is no page fault. We already see that OPT has done better than FIFO, since this same reference to page 2 did cause a page fault under FIFO.

Next, page 4 is referenced. Of the three pages in memory, 2, 3, or 5, which should be replaced? We look into the future, and see that 5 will be referenced first, then 3, then 2, and so since 2 will be referenced *furthest in the future*, it is selected for replacement, and page 4 overwrites it.

By the end, a total of six page faults occur. Of these, five are obligatory, and so the remaining one page fault is what can be directly attributable to OPT. In fact, this is the minimum number of page faults; no policy can possibly do better than OPT. We now have a point of comparison. OPT is

responsible for one page fault, whereas FIFO is responsible for four. FIFO did a lot worse.

In fact, FIFO generally does a lot worse than OPT. Our simple example illustrates this, and while it doesn't prove it (we'd have to come up with a theoretical argument for this), if we were to try a large number of reference strings, we'd see that our example is characteristic of the behavior of the far majority of them (we contrived our sample reference string precisely for this purpose, to produce representative behavior). In general, for most reference strings, FIFO will not do well compared to OPT.

How would we implement OPT in a real system, i.e., how would we determine the optimal page? This would require knowledge of the future, and there is no way to predict, in all situations, whether and when a page will be accessed. It would require knowing how a program will run, i.e., whether it will reach a certain point, and we know from theoretical computer science that this is impossible; the only way to know if a statement will be reached, in a general program, is to actually run it!

Thus, the OPT policy *cannot be implemented in a real system*. However, it can be used in simulations (as in our example). Given a memory reference string, we could apply OPT to it, as having such a string tells us what will happen in the future.

Consequently, simulating how various policies perform given a memory reference string, we could compare them and see which one causes more page faults, which is precisely what we did. By making one of these policies OPT,

we will know what are the least number of page faults. While we can never design a policy that will be as good as OPT, it is still good to know how close the candidate policy is to OPT. If it were, say, within 5% (i.e., it causes 5% more page faults than OPT), then we would say we have a good policy for implementation. In our example, we see that FIFO is far away from OPT.

Keep in mind that we can only make comparisons as we are doing by applying the policies to a specific memory reference string. Perhaps the candidate policy performs well for some, and poorly for others. Thus, it is important to come up with a memory reference string that is indicative of actual programs that will be run on the system. (Finding such a memory string should not be confused with the reference string that we are using, which is extremely unrealistic, used only for expository purposes. An actual reference string would be many millions, or even billions, of references. Consider that for a program that runs for one second, assuming a minimum of one memory reference per instruction, there would be at least ten million references).

In fact, finding one such string is not good enough, as no single string can be indicative of a wide variety of programs. Consequently, the typical approach is to use many memory reference strings, each representing a different program (or type of program). This is called a workload, and a workload that is indicative of typical real workloads (i.e., a set of real programs that run in a real system) is called a *representative workload*.

Thus, we would see how well OPT performs on a representative workload,

and then use our candidate policy on the same representative workload to see how it performs relative to OPT. If it is very close, we know we have a good page replacement policy (but only as good as the degree to which the representative workload is indicative of the actual real workloads that will be used in the system!).

10.10 Least Recently Used Page Replacement

The *LRU* policy selects the page that was “least recently used,” which means used furthest in the past. Why should this be a good candidate page to remove? Because, if the past is a good predictor of the future, then this page will likely not be referenced in the near future. In fact, since all other pages have been used more recently than this least recently used page, it is presumably the best one to remove. That, at least, is the reasoning. But does it make sense?

The key to this argument lies in whether there is locality of reference in the memory reference string. If so, we can use the past as a predictor of the future. Recall that locality of reference means that references cluster in time or space or both. This means that if a page has been referenced recently, it will likely be referenced in the near future. Such a page is an example of the type of pages we do not want to remove. That leaves pages that were not referenced recently as candidates for removal, and the least recently used page is presumably the one most worthy of removal. Note that this only make

sense if the future does in fact behave like the past; otherwise, it doesn't. While there is no guarantee the past predicts the future, if there is locality of reference, it is about the best we can do.

To clarify how LRU works, its behavior on the page reference string is shown in Figure 10.7.

Reference string	2	3	2	1	5	2	4	5	3	2	5	2
LRU	2*	2	2	2	2	2	2	2	3*	3	3	3
7 faults		3*	3	3	5*	5	5	5	5	5	5	5

Figure 10.7: The LRU page replacement policy. After the first three obligatory page faults, page 5 is referenced. The page it will replace is page 3, because page 3 was referenced furthest in the past; the other two pages in physical memory, page 2 and page 1, were referenced more recently.

The first four references are to pages 2, 3, 2 again, and 1, and these will simply fill the first three frames, just as we've seen earlier. These page faults do not cause any page replacement, so up to this point, there is no difference between any of the policies.

The next reference is to page 5. This is an obligatory page fault (because page 5 has never been referenced), but the policy now makes a difference, because a page will have to be replaced. LRU chooses to replace page 3 in the 2nd frame (compare this to FIFO and OPT). To see why LRU makes this selection, consider all the possible choices LRU had. It could have chosen pages 1, 2, or 3 (those are the only pages in physical memory). Consider

each one, and find when in the past they were referenced.

Page 1 was referenced in the previous time unit (just before the reference to page 5), and so it is certainly not a good choice *if there is locality of reference*, which tells us that if a page was recently referenced, it will be referenced again very soon. Page 2 was referenced just before page 1. Page 3 was referenced before that; it is the *least recently used* page. Consequently, it is the choice we make, and so page 5 overwrites page 3, leaving pages 1 and 2 in physical memory.

In the next time unit, page 2 is referenced, and since it is in physical memory, there is no page fault. We see that LRU, like OPT, has done better than FIFO, since this same reference to page 2 did cause a page fault under FIFO.

Next, page 4 is referenced. Of the three pages in memory, 2, 3, or 5, which should be replaced? We look back to the past, and see that 2 was just referenced, then 5, then 1, and so since 1 was referenced *furthest in the past*, it is selected for replacement, and page 4 overwrites it.

By the end, a total of seven page faults occur. Of these, five are obligatory, and so two page faults are directly attributable to LRU. Recall that the number of additional page faults under OPT was one, and under FIFO it was four. LRU came very close to OPT, and did much better than FIFO. And in fact, in general, when there is locality of reference, LRU does very well, approaching the performance of OPT, and performing much better than FIFO.

While LRU generally achieves results that are much better than FIFO, unlike FIFO, it comes at a much higher cost. Consider what is required to determine that a page is the least recently used page. How can we tell which page this is? One way would be to “time-stamp” a page whenever it is accessed, i.e., record its time of access. Then, the least recently used page would be the page with the earliest time stamp.

There are some difficulties with this. We would require a word of memory, enough to store a time stamp for every page in physical memory, which might amount to one for every frame. Secondly, there needs to be some mechanism to read the time from a clock and store it in the word of memory every time a memory operation occurs. Finally, we need a way of finding the earliest time stamp to identify the least recently used page.

10.11 FIFO vs. OPT vs. LRU

Let’s summarize what we’ve learned so far. We have reviewed three page replacement policies: FIFO, OPT, and LRU.

FIFO orders pages by when they were brought in, and replaces them in that order: the first in is the first out. It is the simplest and easy to implement (and to understand), but its rationale for which page should be replaced, the oldest, is weak, and in fact, it generally does not perform well.

OPT selects the page that will be used furthest in the future (or never); this produces the least number of page faults, and thus, is optimal. Since

this selection requires future knowledge, it cannot be implemented in a real system. Thus, OPT is a theoretical policy, which can be used in simulations to tell us the least number of page faults that a particular reference string will cause, and when compared to other policies, we can see how well those policies do relative to the optimal one.

LRU replaces the page that was referenced furthest in the past, and thus least recently used. Its rationale is to use the past as a predictor of the future, which holds if there is locality of reference, i.e., if a reference to a memory address occurred recently, there is a high likelihood that it, or a nearby address, will be referenced soon again. Assuming there is locality of reference, LRU does very well, close to OPT and much better than FIFO. However, its implementation cost is high, as it requires a word of memory per page resident in physical memory to store a time stamp when that page was last referenced, and requires finding the earliest time stamp to identify the least recently used page. This involves considerable hardware, software, and potentially, significant run time overhead. Despite these costs, we are willing to still consider LRU only because of the tremendous time penalty due to accessing the disk when a page fault occurs; avoiding page faults may be able to make these costs acceptable.

Other than OPT being optimal, the other comparisons are general ones: they generally hold true, but not necessarily. Is it possible that FIFO performs as well as OPT? Yes, if it just so happens that the oldest page is also the one that will be referenced furthest in the future. Is it possible that LRU

performs worse than FIFO? Yes, if when comparing the oldest brought in page to the earliest referenced page, the former is less likely to be referenced again. So while in general, $\text{OPT} > \text{LRU} > \text{FIFO}$, we can have $\text{OPT} > \text{FIFO} > \text{LRU}$. We can also have $\text{OPT} \geq \text{LRU} \geq \text{FIFO}$ and $\text{OPT} \geq \text{FIFO} \geq \text{LRU}$. But in general, assuming there is locality of reference (which is normally a safe assumption), $\text{OPT} > \text{LRU} > \text{FIFO}$, with LRU being close to OPT and much better than FIFO.

Given this, do we go with FIFO, which has low implementation costs but does not perform well, or LRU, which performs well but is very costly to implement? Perhaps there is a middle ground?

10.12 Approximating LRU

Consider that our goal is to minimize page faults by selecting a page that has a low likelihood of being accessed, relative to the other resident pages. It doesn't have to be the "best" candidate, as dictated by OPT; it just has to be good enough. This argues for *an approximation* to LRU: select a page that, while it may not be the least recently used, is at least *not* recently used. In other words, find a page that simply hasn't been recently accessed.

How might we identify such a page? Here we need some help from the hardware. Say we had a single bit associated with each page in physical memory such that when a page is referenced, the hardware sets its bit to 1. This is called the *reference bit*, and as we saw in the discussion on page

tables, a page table entry includes such a bit. The page table entry includes an additional bit, called the *modified bit* (sometimes called the dirty bit), which indicates whether the page has been modified.

We can use the reference and modified bits as follows. When a page is brought into physical memory (paged in), its reference bit and modified bits are set to 0 *by the kernel*. When the process to which that page belongs runs and references the page, the reference bit is set to 1 *by the hardware*. If the reference caused the page to be modified, then the hardware also sets the modified bit to 1. By these settings, the kernel is able to tell whether a page has been referenced, and if so, whether it has also been modified. This is valuable information that the kernel will make use of when it handles page faults.

It is critical to make the distinction as to what causes the setting of the reference and modified bits to 0 or 1, whether it is the kernel or the hardware. If we required the kernel to set the reference and modified bits to 1, that would mean that, *for every memory reference, the kernel would need to be invoked*. This would be incredibly expensive, and so the mechanism is, and indeed must be, supported by hardware. As for setting the reference and modified bits to 0, this happens infrequently, e.g., when a page is brought in, which is something that is initiated by the kernel anyway, and so adding an extra instruction to also set the reference and modified bits to 0 is a minimal cost.

Of what use are these settings of the reference and modified bits? Let's

first consider the reference bit. If the reference bit is 1, it tells the kernel that the page was referenced since the time the kernel set it to 0. Imagine that the kernel sets the reference bit of a page to 0 not just when the page is brought in, but does so periodically. Then, when the kernel observes the page's reference bit, if it is 1, it knows whether the page was accessed within that period of time. More importantly, if the bit is 0, it knows that the page was not accessed within that period of time. Consequently, it may be a candidate for page replacement, because it was *not recently accessed*, where “recent” corresponds to whatever period of time the kernel uses.

The modified bit is of value because it tells the kernel whether the page has ever been modified. When a page is brought into the physical memory from the disk, there are actually two copies of that page – one in physical memory and one on the disk – and they are exactly the same. If the page is referenced but not modified, the copies remain the same. If the page is referenced and modified, then the copies diverge. This is a key observation because, when a page is replaced/overwritten, if the page was never modified, it does not need to be saved to the disk because there is already a copy of it on the disk that is the same. However, if the page was modified, then the page must be saved to the disk *before* it is overwritten. Saving a page to the disk is yet another disk access, and we know disk accesses are very expensive; consequently, the modified bit helps to know whether such a disk access is necessary or can be avoided.

These two bits, the reference and modified bits, can be used in a page

replacement algorithm to significantly reduce the number of disk accesses. The reference bit helps determine whether a page was recently accessed, and if so, that page should not be replaced as there may be better candidates (pages that have not been recently accessed, indicated by their reference bits being 0). The modified bit determines whether a page to be replaced must be first rewritten to the disk.

10.13 The CLOCK Algorithm

The *CLOCK algorithm* is a page replacement algorithm that approximates the behavior of the LRU page replacement policy. It does so through the clever use of the reference and modified bits, and works as follows. First, all the frames of physical memory are arranged in a circle. This simply means that we order frames by their number, and that after the highest frame number comes frame number 0. Second, we keep a pointer, called the clock hand, that will refer to the next frame to *consider* for page replacement, i.e., the page in that frame will be the next one to come under consideration for being replaced when room is needed to bring in a new page.

Note that this does not mean that that page will be the next one to definitely be replaced, but only that it will be the next one to be considered, which may or may not result in its selection. If it is not selected, the clock hand will move to the next frame, and the page in that frame will be the next one to be considered. Arranging frames in a circle with a clock

hand that moves around (clockwise, i.e., by sequentially increasing the frame number being pointed to) is why this is called the CLOCK algorithm. This arrangement of frames with a clock hand is shown in Figure 10.8.

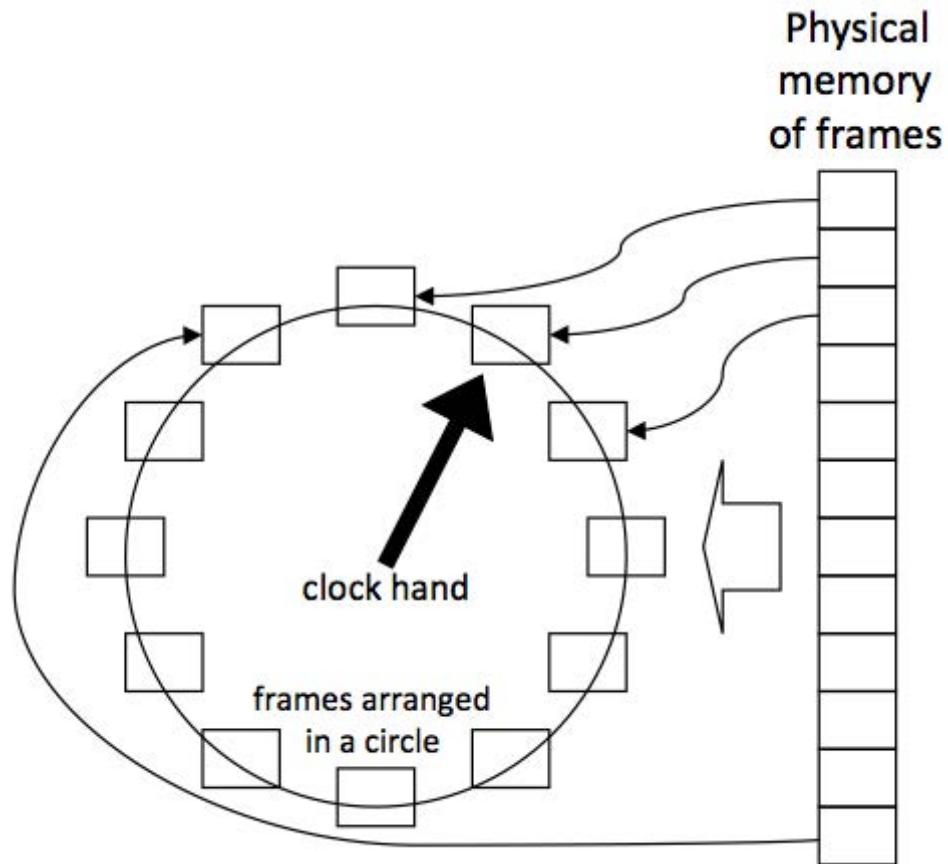


Figure 10.8: The CLOCK algorithm arranges frames in a circle with a pointer, the clock hand, which moves clockwise around the frames.

The CLOCK algorithm can be invoked when a page fault occurs. The kernel handles the page fault, and if there is need to make room for the new page, it runs the CLOCK algorithm, which considers whether the page in

the frame pointed to by the clock hand should be removed. The decision is simple: is the reference bit 0 or not? If it is 0, then this page has not been referenced recently (at least since the time the kernel set the bit to 0; recall that the kernel always set the reference bit to 0, while it is the hardware that sets it to 1). Consequently, the page is removed by overwriting it with the page being brought in.

Before doing so, the page may have to be saved to the disk if it was modified, which is known by checking to see whether the modified bit is set to 1. If it is set to 0, the page was never modified, and so it can be safely overwritten, knowing that a copy of that page is already on the disk and can be used later if and when that page is needed to be brought back into physical memory.

After the completion of these operations, the page table entry of the remove page is no longer valid, and so the valid bit is set to 0. After the new page is brought in, the page table entry for this new page is updated by recording the frame number to which it was brought in, the reference and modified bits are both initialized to 0, and the valid bit is set to 1. Finally, the clock hand is forwarded to the next frame in the circle, and we are done.

When the process is eventually resumed, the instruction that caused the page fault will be retried, but this time no page fault will occur and it can proceed. (This is assuming that between the time that the page was brought in and the time that the process is able to run, the page did not get removed, which is actually a possibility! In fact, there are numerous complications that

we are ignoring, which we will eventually get to, to simplify the discussion.)

If the reference bit is 1 (of the page that is being considered for removal), then this page has been referenced recently (since the time the kernel set the bit to 0). This is not a good page to be removing, and so the clock hand is advanced to the next frame. However, *the reference bit of the page that was just checked is reset to 0*, to effectively begin a new period for this page as to whether it will be kept or removed, i.e., whether it was recently accessed or not, when it eventually comes up for consideration for being replaced. This will occur when the clock hand has managed to go all the way around the circle. Indeed, the average time it takes for the clock hand to go around can be considered the definition of “recent.”

Note that the modified bit is not reset to 0, as it is important to maintain knowledge as to whether the page was ever modified or not. Thus, it is possible that a page can have a reference bit set to 0 and the modified bit set to 1, even though it might seem illogical that these bits indicate that the page was modified but not referenced. It is thus important to be clear on the meaning of the reference bit, which is that the page has not been referenced *since the last time the kernel set it to 0*, and not that the page was never referenced!

Continuing with the algorithm, if the reference bit of the last inspected page was 1 (and we advanced the clock hand), since we have still not identified a page for replacement, the kernel now considers the page being pointed to by the clock hand. The above considerations are now repeated: if the reference

bit of this page is 0, then it is removed, possibly involving a saving of the page to disk if it was modified, and replaced/overwritten, with the corresponding updates to the page table entries; if the reference bit is 1, then it is reset to 0, the clock hand is advanced to the next frame, and the page in this frame is considered for removal. This continues until a page is found for which its reference bit is 0. Notice that since every time the clock hand is advanced, the reference bit of the skipped page is set to 0, eventually, it will encounter a page with the reference bit set to 0. It may take an entire revolution of the clock hand, but it will surely happen.

To clarify how CLOCK works, let's apply its operation to a physical memory of three frames as shown in Figure 10.9, and a sample page reference string consisting of seven page references: 5 9 7 1 9 5 9. Page 1 is underlined to indicate that it is the next page to be referenced, as pages 5, 9, and 7 have been referenced and are already in the physical memory. The frames are organized as in a clock face, in a circle in clockwise order, and with a clock hand pointing to the first frame that contains page 5.

Next to each frame, at the upper left, is a small box that shows the reference bit for the page contained in that frame. When each page was brought in, the kernel would have set the reference bit to 0, but when the process that caused the page fault is resumed to run, it will reference that page and the reference bit will be set to 1 by the hardware.

It is important to not be confused as to whether reference bits are associated with pages or frames, despite that in Figure 10.9, it seems that they

are associated with frames. To be clear, *reference bits are associated with pages*, not frames; each page has its own reference bit, located in the page's page table entry in its page table. However, we show a reference bit next to a frame because a page's reference bit is *irrelevant if that page is not in physical memory*; it is only relevant if the page is in physical memory. But if it is in physical memory, it must be in some frame. So, when we show a reference bit next to a frame, it corresponds to the page that happens to be in that frame, and not the frame itself.

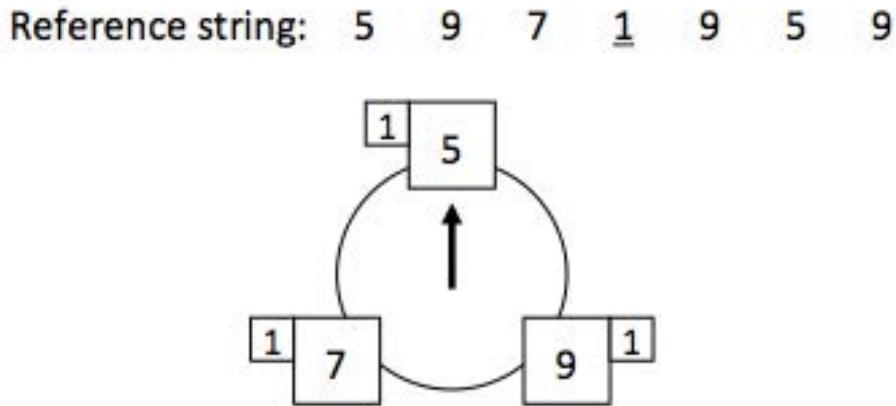


Figure 10.9: A page reference string and a 3-frame physical memory organized in a circle for the CLOCK algorithm. Shown next to each frame, at its upper left, is a bit which corresponds to the reference bit of the page that is in the frame. Pages 5, 9, and 7 have already been referenced, and page 1 is about to be referenced.

The next memory reference is to page 1, which is not in the physical memory. Consequently, a page fault occurs, the kernel is given control to

handle the page fault, and determines that it must remove a page to make room for page 1. The kernel then runs the CLOCK algorithm, which checks the page in the frame pointed to by the clock hand, which is page 5. Since its reference bit is 1, this page has been referenced recently (recent enough such that the reference bit never got turned off after being referenced), and so the reference bit is set to 0 and the clock hand is advanced, as shown in Figure 10.10.

Reference string: 5 9 7 1 9 5 9

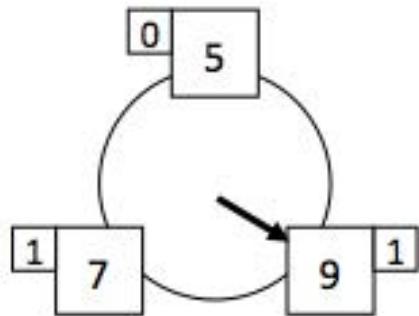


Figure 10.10: Page 5 was skipped because it was recently referenced (given its reference bit was 1), and so the reference bit gets set to 0, and the clock hand is advanced.

The same situation is true for page 9, whose reference bit is 1 (i.e., it is considered “recently used”), and so its reference bit is set to 0 and the clock hand is again advanced, as shown in Figure 10.11.

And again, the same situation for the previous pages is true for page 7, whose reference bit is 1, and so its reference bit is set to 0 and the clock hand

Reference string: 5 9 7 1 9 5 9

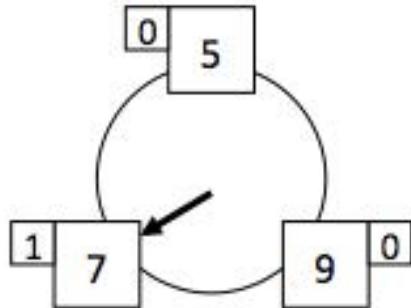


Figure 10.11: Page 9 was skipped because it was recently referenced (given its reference bit was 1), and so the reference bit gets set to 0, and the clock hand is advanced.

is again advanced, as shown in Figure 10.12.

Keep in mind that the kernel has been running throughout to handle the page fault due to the process having referenced page 1. It is still in the middle of running the CLOCK algorithm to find a page worth removing. It has considered every page in physical memory, and did not find one.

The clock hand now points to page 5, having made a complete revolution. When the CLOCK algorithm began running as a result of the current page fault, it began with the clock hand pointing to page 5, but at that time, the reference bit was 1. The implication of this was that page 5 was recently used, and so should be skipped. It was skipped, but in doing so, the reference bit was set to 0, essentially allowing the system to “forget” that the page was recently used.

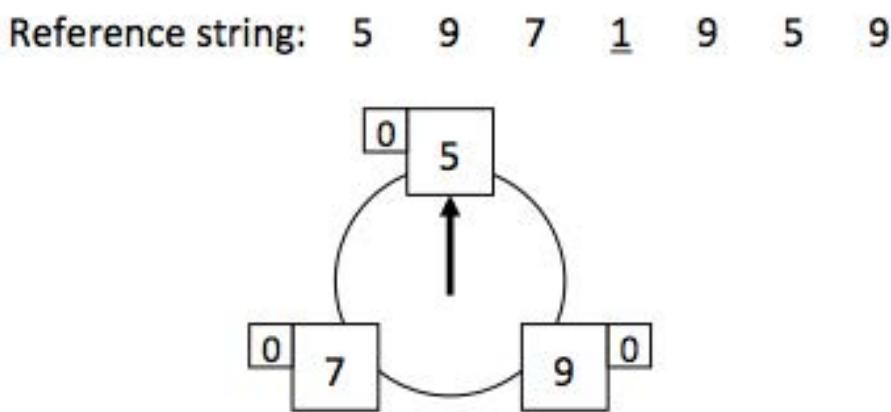


Figure 10.12: Page 7 was skipped because it was recently referenced (given its reference bit was 1), and so the reference bit gets set to 0, and the clock hand is advanced. The clock hand has made a complete revolution, each time setting the reference bit from 1 to 0, and so it is now guaranteed to have found a page with the reference bit set to 0.

We now see why, because if every other page is also considered recently used, there must be some way of defaulting to some more basic criteria in finding a page to replace. The criteria include the use of age, and so CLOCK defaults to FIFO! Page 5 happens to be the earliest page brought in (ahead of pages 9 and 7), and so, given no way to distinguish them based on their time of usage, their age is used.

Using age, in the absence of any other criteria, may be the best we can do. There is some rationale, weak as it may be, that pages that have been brought in later will have been more recently used. On the positive side, it is at least simple to implement. Regardless, this situation (that CLOCK defaults to FIFO) should rarely happen, as in a real system, the number of frames will be in the millions, and so the likelihood that every single page was recently used should be low.

Continuing with the algorithm: since the clock hand points to a page whose reference bit is 0, page 5 in our example, this page is selected for removal. The clock hand is advanced, pointing to page 9. The CLOCK algorithm is now done; it identified a page to be replaced. The clock hand was advanced since we always want it to point to the page that should *next* be considered for removal, whenever the CLOCK algorithm is invoked in the future.

While the CLOCK algorithm is done, the kernel still has work to do, as it is still not done in handling the page fault. It makes a disk request to actually bring in page 1 into physical memory, specifically into the frame that was

occupied by page 5. The page table entry for page 5 (not shown) is updated by setting its valid bit to 0, since page 5 is no longer in physical memory. The kernel initializes the page table entry for page 1 (also not shown) by setting the frame field to the first frame (frame 0) since that is the frame that page 1 occupies, the reference bit is set to 0, and the valid bit is set to 1.

The page fault is now considered “handled,” and the kernel can return control to the process that caused the page fault, which can be resumed. That process will reissue the instruction that made a reference to page 1, but this time, no page fault will occur. In fact, the page will be referenced, which will succeed, causing the hardware to set the reference bit to 1, as shown in Figure 10.13.

Reference string: 5 9 7 1 9 5 9

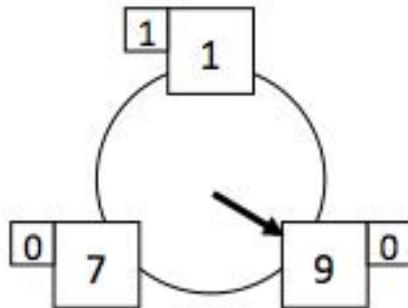


Figure 10.13: Page 1 replaced page 5, and its reference bit was initialized to 0. However, once the process that caused the page fault is resumed, the page will be referenced, and the reference bit will be set to 1 by the hardware.

We have glossed over, even ignored, a few complications, and so before

continuing with the example, let's examine them. First, we have ignored whether the page being replaced, page 5, was modified, which would be indicated by the modified bit equaling 1. In this case, page 5 would have to be written out to the disk before page 1 is brought in.

Second, we have glossed over the fact that a disk request takes significant time. It would be unreasonable for the kernel to simply wait until the page was brought in (and worse, if the page being replaced was modified, waiting until that page was written back to disk before bringing the new page in) before returning and allowing a process to run. But the process that caused the page fault cannot run until all the disk requests complete (if it did, it would simply cause another page fault) so it still cannot run. However, the kernel can schedule other processes that are in the ready state, which is what it would do. If there were no other process that was ready to run, then the kernel would indeed just wait (until the disk generates an interrupt, which would then cause the kernel to handle that interrupt, and if the interrupt was due to the completion of having brought in a page, the corresponding waiting process would be woken up, and since it is now ready to run, the kernel would allow it run)).

Continuing with our example: page 9 is now referenced. Page 9 is in the physical memory, so there is *no page fault*. However, its reference bit, which is currently 0, is set to 1, as shown in Figure 10.14. This is done automatically, *by the hardware*. Note that the hardware knows nothing about the CLOCK algorithm, that pages are organized in a circle, that there is a clock hand,

etc. However, it does know about a page's reference bit, and sets it whenever that page is referenced (the same goes for the modified bit). Since it knows nothing about the clock hand (which is only updated when the CLOCK algorithm runs, by the kernel), the clock remains where it was, pointing to page 9.

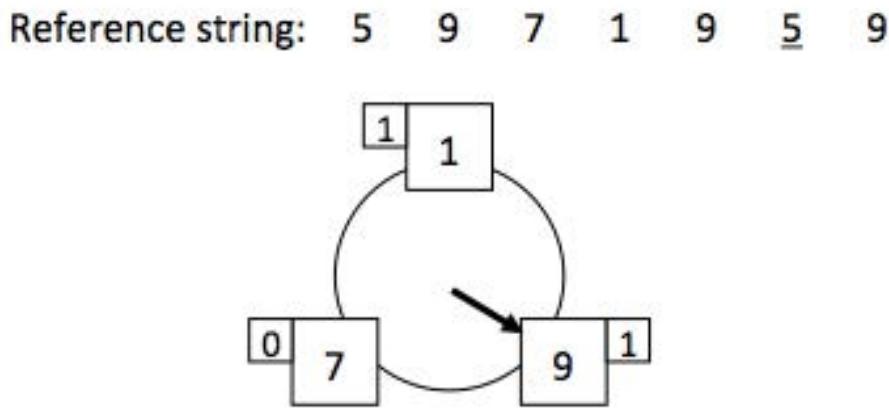


Figure 10.14: The state of the physical memory after referencing page 9.

The kernel relies on the hardware setting the reference bit, so that when the kernel observes that a page's reference bit is 1, it knows that the page was referenced “recently,” i.e., between the time the kernel had set that page's reference bit to 0 and the current time. Thus, the reference bit effectively is a *one-bit time stamp*, where time is divided into the “recent past” and the “not-recent past.” Since the kernel can determine when it sets the reference bit to 0, it essentially defines what corresponds to the “recent past.” When the kernel uses the CLOCK algorithm for page replacement, it defines the

recent past as the time it takes for the clock hand to make one revolution around the clock.

Next, page 5 is referenced. Since page 5 is not in the physical memory, a page fault occurs. The kernel handles the page fault, and runs the CLOCK algorithm to determine which page to replace to make room for page 5. The clock hand is pointing to page 9, but since its reference bit is 1 (because it was recently referenced), this page is skipped over and the hand is advanced to point to page 7. The reference bit for page 9 is also reset to 0, as shown in Figure 10.15.

Reference string: 5 9 7 1 9 5 9

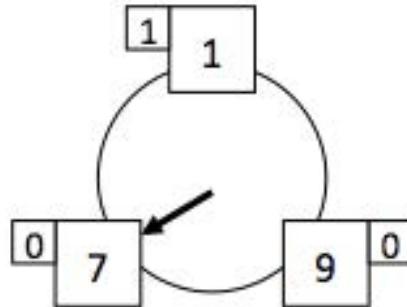


Figure 10.15: Page 9 was skipped because it was recently referenced (given its reference bit was 1), and so the reference bit gets set to 0, and the clock hand is advanced.

Since page 7 has its reference bit equal to 0, it is considered not recently used, and is selected for replacement. Page 5 will be brought in (after page 7 is written to disk if its modified bit was 1), overwriting page 7, the reference

bit will then be initialized to 0, and the clock hand will be advanced. This is all done by the kernel, which can then allow the process that caused the page fault to resume. When the process starts running again, it reissues the instruction that caused the page fault, which will reference page 5, but this time page 5 is in memory, and so no page fault occurs. Given the reference, the hardware sets the reference bit for page 5 to 1, as shown in Figure 10.16.

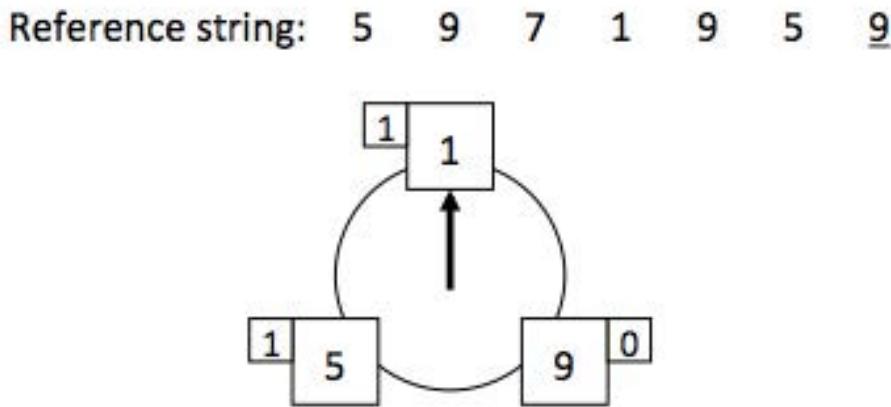


Figure 10.16: Page 5 replaced page 7, its reference bit was initialized to 0, and the clock hand advanced, and the process resumed, referencing page 5, which is why the reference bit is set to 1.

Finally, page 9 is referenced. Since page 9 is in physical memory, no page fault occurs, and the hardware will set the reference bit for page 9 to 1, as shown in Figure 10.17.

Every page in the reference string, at least the part shown, has been referenced. Notice that the clock hand remains pointing to page 1. In fact,

Reference string: 5 9 7 1 9 5 9

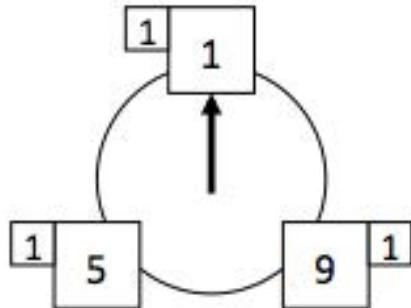


Figure 10.17: The state of the physical memory after referencing page 9.

looking at the reference string, relative to the end point, page 1 was the page referenced furthest in the past. Consequently, if there were more references to be made, the next page fault would cause page 1 to be replaced. However, it would not happen upon first inspection of what the clock hand is pointing to, because page 1 has its reference bit set to 1. The CLOCK algorithm would behave just like we started in this example, where the clock hand will move all the way around, setting the reference bits to 0, until it again encounters page 1, but this time with its reference bit set to 0. It was selected because it was the oldest page, which in this case, coincides with it also being the least recently used page.

In summary, the value of the CLOCK algorithm is that it performs fairly well, it is simple to implement, and while it does require hardware support, the amount is minimal. It performs well because it is based on LRU; it is a

rough approximation of it by removing a not recently used page, rather than the least recently used of LRU. It is simple to implement because it organizes frames in a circle according to their number, using a clock hand to track the next candidate page to remove. This also allows CLOCK to default to FIFO if all pages are considered recently used. It requires hardware-supported reference bits, one per page in physical memory, such that when a page is accessed, the reference bit is set to 1, which allows the kernel to determine recent vs. not recent access. Most modern systems provide this support. The reference bit acts as a one-bit time stamp, separating pages into those that were recently used vs. not recently used.

Modern operating systems use an advanced version of the CLOCK algorithm to implement page replacement. Some of the more advanced features include:

- Maintaining a “pool” of free frames so that when a page fault occurs, there is no need to search on the spot for a frame because the search took place earlier. This is implemented by a daemon, a process that only runs in the kernel, and is periodically woken up to fill the pool. It may make use of a second clock hand that sweeps ahead of the primary hand by identifying the good candidates that become part of the pool, leaving the primary hand to be pointing to the frame containing a page that is to be removed next.

Note that the reference bit of that page is always checked, just in case

the page was referenced between the time it was identified by a good candidate to be removed (and thus part of the pool) and the current time. In this case, it would be skipped over (setting its reference bit to 0) and the primary clock hand would be advanced to the next page identified as a good candidate by the second clock hand.

- Bringing in additional pages to the one that is actually needed due to a page fault. These additional pages are related in that there is a good chance they will be referenced. For example, if page 7 is referenced and causes a page fault, both page 7 and page 8 (and perhaps even others that logically come after page 8) will be brought in, the idea being that if there is spatial locality of reference, page 8 will likely be referenced after page 7.

This addresses an issue that might have seemed mysterious in our description of the CLOCK algorithm, which we can now address. When a page is brought in, the kernel sets the reference bit to 0. This may have seemed of minimal consequence given that, once the page is brought in and the process that caused the page fault is resumed, the page will be referenced and the reference bit will be set to 1 by the hardware.

So, why bother initializing it to 0? Now we can see why. If multiple pages are brought in, some only based on a good guess that they will be referenced, it is important that their reference bits be set to 0. This way, those that are not referenced will be known as pages that have

not been used (because their reference bits will have remained being 0), and if they remain in the system for a long time, they will be favored for replacement over others that are being referenced.

We end our discussion of the CLOCK algorithm with the following observation. The speed at which the clock hand moves has meaning. It is an indicator of how much demand there is on the physical memory. The faster the clock hand moves, the greater is the page fault rate, and the slower the system will operate because of all the disk accesses. At some point, there is so much demand on the physical memory that all processes suffer, which is clearly not a good situation.

This leads us to consider the following strategy: when there is high enough demand on physical memory, reduce the number of processes that are competing for physical memory. This is done by identifying a group of processes that will be allowed to be in physical memory, at least in part, while all the others are made to wait by keeping their entire memories on the disk, i.e., none of their parts will be in physical memory. The idea is that this smaller group may be able to use the physical memory more effectively than if they were all competing simultaneously. For fairness, and after some interval of time, those that were on disk can be given a chance by making them part of the group while those that were part of the group for a significant time can be removed.

10.14 The Resident Set

To elaborate on this idea, at any point in time, a process will have some of its pages in physical memory (as not all will be needed; indeed, those that have not been referenced ever or for a long time need not be in memory, assuming locality of reference). This set of pages is called the *resident set* of a process. One can imagine that there is some “optimal” resident set, i.e., those pages that a process will be referencing soon and so should be in memory (to avoid page faults), while those that will not be referenced soon need not be in memory. We’ll make the term “soon” more precise below.

To arrive at this optimal resident set, we must ask the following questions:

- Which pages should be part of the resident set? As discussed above, those that will be needed soon should be part of it, while those that won’t be needed soon need not.
- How big should the resident set be? From the process’s point of view, the bigger the better. On the other hand, the bigger it is, the less space there is for other processes. Consequently, this question depends on the other processes, and specifically, their resident sets.
- If there is a page fault for a process, when the kernel brings in the needed page (to make it part of the process’s resident set), should the page being replaced come from that process’s resident set, or from the resident set of another process?

10.15 Local vs. Global Page Replacement Policies

This last question leads us to refine our categorization of page replacement algorithms into local vs. global.

A *local* page replacement policy is one where the page brought in for a process replaces one of its own resident pages. This has the important property of isolation of process memories, such that the memory behavior of one process does not affect that of any other. If one process begins to generate page faults at a high rate (it begins to repeatedly reference pages that are not in its resident set), there will be minimal effect on other processes in that their resident sets are not affected, i.e., their pages are not removed to make room for the process that is page faulting.

On the negative side, a local policy can be inefficient in its use of physical memory. Consider a process that requires a much larger resident set than it has to run efficiently, without page faulting. If there are other processes with resident sets that have pages that are relatively unused, ideally, those pages can be replaced and the frames they occupy could be given to the process that needs to enlarge its resident set. But, with a local policy where there is strict isolation, this would not happen. One can imagine relaxing this rule (which is what is done in practice), but then the isolation property begins to break down.

A *global* page replacement policy is one where the page brought in for a

process can replace any page in physical memory, regardless of which process that page belongs to. All the page replacement policies we have considered so far (FIFO, OPT, LRU, CLOCK) are global policies. (We will study a local policy below.) Global policies lack the isolation property, and so allow processes to negatively affect the memory behavior of other process. If one process begins to generate page faults at a high rate, pages from the resident sets of other processes can be removed, which may cause *those* processes to begin page faulting (because the pages they expected to be in their resident sets are no longer present).

On the positive side, a global policy can be more efficient in its use of physical memory than a local policy. If a process that requires a much larger resident set that it has to run efficiently, and there are other processes with resident sets that have pages that are relatively unused, those pages can be replaced and the frames they occupy could be given to the process that needs to enlarge its resident set. Ideally, every process's resident set would grow or shrink based on their current memory needs. However, if the demand is too high, all the processes will suffer, and there will be a continuous taking of memory from each other, which may lead to an ever-increasing overall page fault rate.

10.16 Multiprogramming Level

To clarify this issue, consider the concept of *multiprogramming level*, which simply corresponds to the number of processes that are in physical memory. A process is in physical memory if at least one of its pages is in physical memory, i.e., it has a resident set size greater than 0.

Recall from Chapter 3 where we discussed the importance of keeping multiple processes in memory so that, when there is a context switch, there is some other process to run, which promotes the illusion of simultaneous progress. If only one process were allowed to be resident in memory, then the time for a context switch would be overwhelming (tens or hundreds of milliseconds to swap out the memory of the current process and swap in the next one, rather than the few microseconds needed to save and restore the CPU context). So, in the most general terms, we want the multiprogramming level to be as high as possible.

We also want to minimize *overhead*, which roughly corresponds to any work done by the kernel. As important as any of the work that is done to support and maintain the system (and thus processes), what users want is for their programs to run, and not necessarily kernel code. Even if they realize that the kernel is there to help them, they will only tolerate it running as long as it does not get in the way (at least perceptibly) of running their applications! Examples of such overhead are all the mechanisms and policies we've encountered: context switching, CPU scheduling, logical or virtual to

physical address translation, page fault handling, page replacement, etc. It also includes any waiting for resources, such as waiting for the disk to bring a page in.

To minimize overhead, we want to maximize *CPU utilization*, which we define as the fraction of time that the CPU is running code that is strictly part of user programs. If we observe the CPU activity over some interval, and during this interval, K is the time spent in the kernel and U is the time spent running user code (total interval length is $K+U$), then CPU utilization = $U/(K+U)$.

There is a relationship between CPU utilization and multiprogramming level, which is shown in Figure 10.18. If there are no processes in physical memory, then no user program code is running, and the CPU utilization is 0%. If there is one process, the CPU utilization will go up; with two processes, up a little more; etc.

Why does the CPU utilization increase gradually, rather than immediately jumping to 100% when there is at least one single process in physical memory able to run? Because a process does not run user program code exclusively; the process will make system calls (e.g., file read requests), and may even block during those system calls. If there are two processes, there will still be systems calls being made and blocking, but with two processes there is a greater chance that there is “useful” work to do if one of them blocks. Consequently, we expect that, as the multiprogramming level increases, the CPU utilization will increase, just as shown in Figure 10.18.

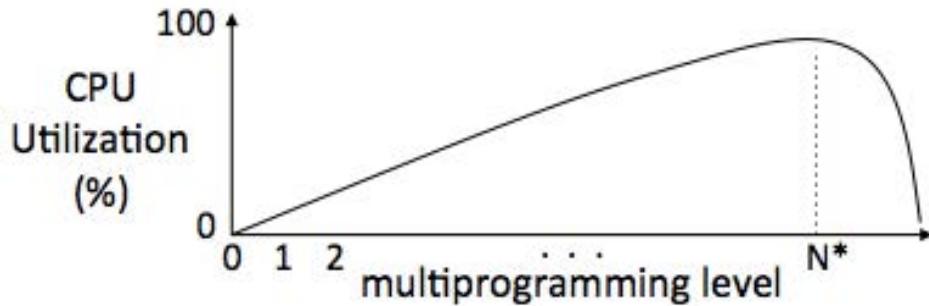


Figure 10.18: CPU utilization vs. multiprogramming level.

10.17 Thrashing

However, there comes a point when there are just too many processes in physical memory. Consider the following extreme situation. Say there is a large number of processes, and each process has just a single one of its pages in physical memory. When one of them runs, when it references any other of its pages (not in physical memory), it will generate a page fault. To bring in that page, an existing page must be replaced. But that means that some other process will lose its sole page in physical memory, and cause that process to page fault (when it runs). And that will cause some other process's page to be removed.

This chain reaction of one page fault causing another, which causes another, which then causes another, etc., is called *thrashing*, making the CPU utilization come crashing down, as most processes will be waiting for their pages to be brought in to physical memory.

There is some optimal value for the multiprogramming level, call it N^* ,

which causes the CPU utilization to reach its maximum value. Finding this value is practically impossible, as it depends greatly on the particular set of processes that are resident, and their individual memory behaviors, which are constantly changing. Some will require larger resident sets than others, and each of their resident set sizes will change over time. While we can't know N^* , we may be able to approximate it, and if only we can operate at a multiprogramming level just before it (based on the shape of the curve in Figure 10.18, we can avoid thrashing).

One way to approximate knowledge of N^* would be to periodically measure the CPU utilization, and when we notice that it is beginning to plateau (i.e., an increase in multiprogramming level does not generate an expected increase in CPU utilization), we are getting near N^* . Or, we might keep a running measure of the page fault rate (i.e., page faults over time, where we measure the number of page faults that occurred over the last time interval since the last measurement), and when we notice that it begins to increase rapidly, we are probably operating just beyond N^* .

10.18 Swapping

Getting near (or, less desirably, having gone just beyond) N^* would trigger the kernel to reduce the multiprogramming level, by removing the entire resident sets of one or more processes. This amounts to removing all of some process's pages in physical memory, and observing the effect on CPU

utilization; this may require removing more than one process. When all the pages of a process are removed from physical memory, this is called *swapping*. Swapping *out* a process corresponds to moving all its pages to disk (which requires only copying out those pages that were modified, as the ones not modified are already on the disk), and swapping in a process corresponds to moving from disk to physical memory all the pages that made up the resident set of a process before it was swapped out.

10.19 The Working Set

We are now ready to address the question as to what should comprise the resident set. It should be the set of pages that are likely to be referenced soon, and that result in a low page fault rate. Removing just one will lead to that process generating many page faults. Consequently, if all the pages that meet this criteria cannot be part of the resident set, then the entire process should be swapped out, as that process will not be able to run efficiently without all of these pages. At the very least, this will free up physical memory so that the needed resident sets of the other processes will comfortably fit in physical memory.

Which pages are likely to be referenced soon? From our earlier discussion on LRU, we know that if a process's memory reference string exhibits locality of reference, we can use the past to predict the future as to which pages will be needed soon. We just have to determine how to define "soon."

The working set is defined by the function $W(t, \Delta)$, which is the set of pages that have been referenced during a “window” of time Δ relative to the current time t , i.e., all (and only) those pages that are referenced between the times $t - \Delta$ and t , as shown in Figure 10.19. The time interval Δ is “process time” and not real time. By process time, we mean the time during which the process is actually running; process time only advances when the process is executing. Thus, the input of this function are the current time and the window as to how far back in the process’s history of memory references to consider, and the output is the set of all pages referenced during that window of time.

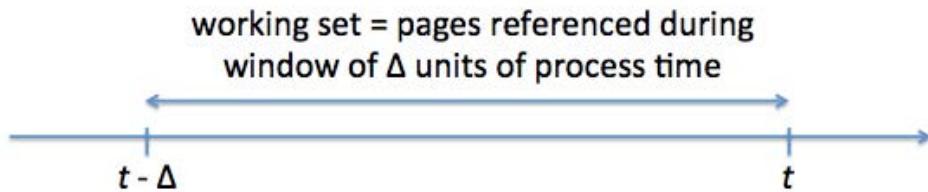


Figure 10.19: The working set.

If the process’s memory reference string exhibits locality of reference, we can expect the pages in the working set to be the ones that will be referenced “soon.” Since time is always marching on, and thus the window is always moving forward (as the process executes), the working set may change, some pages dropping out and new ones being added. The ones that drop out are the ones that were referenced earlier than $t - \Delta$. The ones that are added are the ones that caused page faults during the window $(t - \Delta, t)$.

The key idea behind the working set is that the pages it contains must be a subset of the resident set (ideally the same, but can be a proper subset; as long as the working set is in physical memory, that is what is important). If the working set cannot be kept in physical memory in its entirety, then the process should be swapped out. It just doesn't pay to keep a portion of the working set, as this will result in many page faults (of those pages that the working set indicate will be needed). Keeping pages in addition to the working set will not harm that process, but will waste memory that might be better used by other processes (for their working sets). Thus, the goal is to make the resident set equal to the working set.

One complication is that a process's working set is not static; as time goes on, it may grow or shrink. If it needs to grow (which will happen if there is a page fault while none of the existing pages dropped out), we need to consider whether there is room in physical memory, i.e., a free frame, to hold the incoming page. If there is, the page is brought in and the working set has expanded. If there isn't, this implies that the process needs more memory than there is. If a page from another process (part of that process's working set) is removed, that would harm that process. So, if there isn't enough memory, the process is swapped out, and all the frames containing the pages its resident set are freed and become available for other processes. We now see that this is how free frames become available.

We can now summarize how the working set (WS) page replacement policy works. When the system begins, all frames are part of a free pool that

can be allocated to pages of processes. When a process runs, it will reference its memory and will generate page faults. As long as there is a frame in the free pool, that frame is allocated for the incoming page. Furthermore, that page becomes part of the process's working set and its resident set. As the process runs, when a page in its resident set falls out of the working set (because its last time of reference is prior to $t-\Delta$), the frame containing that page becomes part of the free pool.

The free frame pool will grow and shrink over time; it will grow when a frame is returned due to the page it contains falling out of a working set, and it will shrink when a frame is allocated to store a page due to a page fault. When a page falls out of the working set, it is still part of the process's resident set since it is still in physical memory. This can be used to advantage: if the process happens to reference this page, it will be added back to the process's working set, and this will happen essentially "for free" since no page fault occurs (because the page is still in physical memory). However, the frame that contains it must be removed from the free pool to prevent it from being used for another page.

The optimization leads to even greater savings when one considers pages that have been modified (which will have their modified bits set to 1). When a frame is returned to the free pool, if it contains a page that was modified, that page will need to be written out to disk. However, the disk request can be delayed until the frame is actually needed for another page. If the process with the resident set that contain the modified page runs and references the

page, that page moves back to its working set. If it is then modified, it will eventually have to be saved to disk, and so whatever version of the page was previously saved is useless. By having had delayed the disk request, that disk request is actually avoided without penalty.

We can see that WS is a *local* page replacement policy. Frames containing pages of other processes are never taken away from those processes, and so there is strong isolation between processes. A process that begins page faulting at a high rate will not affect other processes (at least in terms of physical memory allocation and usage, but there may be secondary effects, such as the disk becoming overly busy because of the process page faulting at a high rate, causing disk requests by other processes, for whatever reason, e.g., file accesses, to be delayed). WS is essentially LRU, but taking into account that processes do not operate in a vacuum but rather share the physical memory, and thus there is a need for managing the physical memory allocation amongst all the processes and avoiding thrashing, which is what WS provides. Consequently, WS seems to be the best page replacement algorithm we've encountered (barring OPT, which exists only in theory). There are, however, difficulties with WS that we have glossed over, and must now address.

The first difficulty is determining how a page “falls out” of the working set. We know that a page becomes part of the working set when it is brought in (due to the page fault that was caused by referencing it). But how do we know when the page is no longer part of the working set? We need to know

its time of last reference, which is then compared to $t-\Delta$. This requires a way of time stamping a page when it is referenced (and the time stamp must be in “process time,” which only advances when the process runs, rather than real time, which is always advancing). This would have to be done by the hardware, and would require a word per frame of memory to store the time stamps.

Furthermore, as a process runs, at every time step, the hardware would need to check whether the page with the earliest time stamp has fallen out of the working set, i.e., whether the earliest time stamp has become smaller than $t-\Delta$. The page with the earliest time stamp may be constantly changing, as with every reference, if that reference is to the page with the earliest time stamp, it now becomes the one with the latest time stamp, and some other page is now the one with the earliest time stamp. This implies the maintenance of an ordered list where, every time a page is referenced, that page is moved to the head of the list; whatever page is at the end of the list has the earliest time stamp. This requires a great deal of mechanism, all of which needs to be implemented in hardware (as it would be unreasonable for the kernel to run at every time step to do all this work).

Another problem lies in the value of Δ ; what should its value be? The smaller it is, the smaller the working set will tend to be, and the greater the likelihood that pages that need to be in memory are missed by the working set. On the other hand, the larger it is, the larger the working set will tend to be, and the greater the likelihood that pages that don’t need to be in

memory remain as part of the working set. There is no way to theoretically determine Δ , as it depends on the process's memory reference string and other factors.

The best we may be able to do is to determine Δ experimentally. Select several values for Δ , and run a system with a representative workload (a set of processes that are similar, or will at least have memory behaviors that are similar, to what will typically exist on a system) for some period of time, keeping track of the total number of page faults. Presumably, each different value of Δ will produce a different total number of page faults; the one that produces the smallest will be the best. Clearly, this is highly dependent on the workload, and so there may need to be continuous adjustments. One can imagine adjusting Δ on the fly, systematically increasing and decreasing it and observing the resulting number of page faults.

If these difficulties sound like they will not be easy to deal with, they are, and it is why WS is not used, at least in its pure form as described above. However, just as there are approximations of LRU such as CLOCK, there are approximations of WS that can be implemented, and that attain benefits that are similar to the benefits of WS over LRU, i.e., local vs. global policy, overall physical memory management given many processes.

10.20 Summary

Virtual memory is the logical consequence of logical memory. Logical memory provides an abstraction of memory structured as segments that could be independently addressed, could have different characteristics, and could be shared (or not) between processes. While segments are good from the user's point of view in logically organizing the process's memory, physical memory allocation is simplified if segments are further broken down into equal-sized pages, with the physical memory broken down into same-sized frames so that any page will fit in any frame.

The key observation in going from logical memory to virtual memory is that not all the pages need to be in physical memory. This allows for a virtual memory that is much larger than the physical memory, through the use of a disk to hold the entire logical memory, bringing in only pages (or even entire segments) that are needed at the moment.

To realize virtual memory, a mechanism is needed to identify when a memory reference tries to access a page that is not in memory; this mechanism, implemented in hardware, is the page fault. A page fault causes the kernel to get control, which can then handle the fault by bringing in the needed page from the disk to physical memory, and updating the page table entry to now indicate the page is in physical memory, which frame it is in, so that when the page is referenced again the access will succeed rather than resulting in a page fault.

Eventually, the physical memory becomes entirely filled with pages of various processes, and so there must be a way of replacing pages when new ones need to be brought in. This results in the need for page replacement policies, which determine which pages to replace, with the goal being to remove pages that will no longer be needed to minimize future page faults. Of the various policies, FIFO removes the oldest page, OPT removes the page that will be used furthest in the future, and LRU removes the least recently used page. FIFO is simple to implement but does not work well (as the oldest page may still be a page that will be soon referenced), OPT is a theoretical policy which provides an upper bound on performance, and LRU works well if there is locality of reference, a reasonable assumption, though is costly to implement. The CLOCK algorithm is an approximation of LRU, which makes use of a reference bit per page, essentially a one-bit time stamp, to categorize pages as either recently used and not (as opposed to least) recently used.

A more careful approach to managing the resident set, those pages that are in physical memory, of a process is to seek that it includes the process's working set. The working set is comprised of those pages of a process that have been referenced in the last delta time units in process time. Assuming locality of reference, these are the pages that are expected to be referenced soon, and are thus worth keeping in physical memory. This leads to the WS page replacement policy based on the working set, which, when a new page must be brought in, that page will replace a page in the process's resident

set that is no longer part of the working set. If no such page is available and there are no free frames in physical memory, the process is swapped out, moving all the pages of the process to disk, and at some future point, the process will be swapped back in.

An important distinction between WS and all the other policies is that WS is a local policy, whereas all the others we considered are global policies. In a global policy, a page chosen to be replaced may come from the resident set of any process, whereas in a local policy, a page brought in for a process will replace a page that is part of that process's resident set, thus supporting the desirable property of isolation of memory behavior between processes. While WS has many desirable properties, it is overly costly to implement, though just like CLOCK is an approximation of LRU, there are approximations of WS that can be implemented.

10.21 Exercises

1. How does memory structured as segments and pages allow partitioning of memory for convenient allocation?**
2. How does memory structured as segments and pages allow reorganizing of memory for convenient usage?**
3. What is meant by “convenient” in the above questions?***
4. What is meant by a “logically” organized memory – how is the word

“logical” being used?***

5. One of the implications of a logically organized memory based on segments and pages is that not all the pieces (segments or pages) need to be in memory: precisely what is it that allows us to make this implication?***
6. If not all pieces are in memory, where are they located (i.e., those not in memory)?*
7. What are the factors that support the illusion that there is much more memory, and why?**
8. To create this illusion, what mechanisms are needed?**
9. How is a “logical memory” different from a “virtual memory”?*
10. How does a virtual memory present the illusion of a large memory (and “large” relative to what)?**
11. What is the difference between a “logical address space” and “virtual address space”?*
12. In Figure 10.1, why is there a sequential one-to-one mapping between virtual memory pages and page table entries?**
13. Why is there NOT a sequential one-to-one mapping between page table entries and physical memory frames?**

14. Are all the pages of virtual memory in frames of physical memory?* If not, where are they?
15. How do we keep track of how to find pages of virtual memory that are in physical memory?*
16. How do we keep track of how to find pages of virtual memory that are not in physical memory?***
17. Figure 10.2 shows sample contents of a page table entry: can you explain the meaning of each field?*
18. If the valid bit is off, do the other fields contain useful data?*
19. If not, can you think of a use for the other fields (if the valid bit is off)?***
20. What is a page fault?
21. At what point during address translation is it determined that a page fault is to occur?
22. Is a page fault determined by software (e.g., the kernel) or hardware?*
23. What happens in hardware when a page fault is detected?** ... what happens in software?***
24. What does it mean to “retry” an instruction, and why is it necessary after a page fault?**

25. What are the fields of a virtual address for a virtual memory that is segmented and paged?
26. What is a segment fault, and how is it different from a segmentation violation?**
27. How is a segment fault different from a page fault?*
28. Why are page faults expensive?
29. How does the expense of page faults compare to the expense of TLB misses?**
30. Can you think of any relationship between the time to resolve a page fault and the quantum time?***
31. What is the Principle of Locality (of reference)?*
32. What is meant by “thrashing”?*
33. Why should we expect programs to exhibit locality?***
34. What is meant by “page replacement”?
35. Why is page replacement considered policy?*
36. How does the FIFO page replacement policy work? ... OPT? ... LRU?
What are the pros/cons of each?*
37. What is the key metric of performance for a page replacement policy?**

38. Why doesn't FIFO generally perform well?*
39. Why does OPT perform as best as possible?**
40. Can FIFO ever perform as well as OPT?*
41. Under what conditions will LRU perform well?
42. Under what conditions will LRU not perform well?*
43. What makes LRU so costly to implement?**
44. What is a "reference string"?
45. Do you expect the references in a small (say, 10-100 references) sequence in a reference string to be all the same: why or why not?***
46. Can you explain the FIFO example (each entry in the physical memory frames) in Figure 10.5?*
47. What is meant by an obligatory page fault?
48. In Figure 10.5, why does FIFO incur "9 page faults," and how many of these are obligatory?*
49. Why is removing the oldest page not generally the best?*
50. Can you explain the OPT example (each entry in the physical memory frames) in Figure 10.6?*

51. In Figure 10.6, why does OPT incur “6 page faults,” and how many of these are obligatory?*
52. Why is OPT not realistic?
53. If it is not realistic, why study it?*
54. In what way does OPT do better than FIFO, and why?*
55. Is OPT always better than FIFO?*
56. Can you explain the LRU example (each entry in the physical memory frames) in Figure 10.7?*
57. In Figure 10.7, why does LRU incur “7 page faults,” and how many of these are obligatory?*
58. Was there any locality in this example?**
59. Why does LRU require hardware support, and what does this support correspond to in the example?***
60. Can you explain and justify the inequality “ $\text{OPT} \geq \text{LRU}$ (assuming locality) $\geq \text{FIFO}$ ”?***
61. What is the goal of the CLOCK algorithm?*
62. What is the difference between “least recently used” and “not recently used”?**

63. What is the purpose of the reference bit?
64. The function of the reference bit in the CLOCK algorithm is most similar to what in LRU?***
65. In the CLOCK algorithm, what happens if a memory reference results in a page fault?
66. What happens if a memory reference does not result in a page fault?*
67. What is the purpose of the “hand” in the CLOCK algorithm?*
68. If a frame has been modified, why must it be written to the disk?*
69. In Figure 10.9, can you explain the contents of each of the boxes (including, why do all the little boxes have 1’s), and how configuration was reached given the reference string?**
70. In Figure 10.10, which frame is eligible to hold page 1?*
71. Can you explain what happened to arrive at the configuration in Figure 10.11?
72. In Figure 10.13, why is page 1 placed in the 1st frame?*
73. In Figure 10.13, what caused the reference bit for page 1 to be set to 1, and why was this done by the hardware rather than the kernel?*
74. Can you explain the configurations in Figure 10.15 and Figure 10.16?

75. In Figure 10.17, why is the hand pointing to page 1 while page 9's reference bit was set from 0 to 1?
76. What is the resident set?*
77. What's the difference between local replacement and global replacement policies, and what are the pros/cons of each?**
78. What is meant by isolation vs. efficiency?*
79. What is the multiprogramming level?*
80. What does the graph Figure 10.18 show?**
81. At what point does thrashing occur and why?*
82. What is the working set, and why should the resident set contain it?**
83. What should be done if the working set cannot be kept in memory in its entirety, and why?**
84. Why is the working set a local replacement policy, and what is the significance of this?*
85. What does it mean to time-stamp pages?**
86. Why must pages be time-stamped in the working set?**
87. How should the working set parameter Δ be determined?***

88. How does the working set compare to CLOCK: what are the pros/-cons?**

Chapter 11

File Systems

In addition to the abstractions we've covered so far, processes and virtual memory, we need an abstraction for long-term storage, which is provided by the file system. A *file* is a logical unit of storage, i.e., a container of data. The data it contains is accessed by invoking the file's name, which it is given when it is created, and by specifying a region within the file where the data is located. A *file system* is simply a repository of files, allowing for their storage and retrieval, as shown in Figure 11.1

In addition, the file system provides a user interface that supports a name space for the files and access control, i.e., how they are to be shared between users. The file system also provides management of the underlying storage of the files, with one goal being persistence, i.e., that once a file is saved, it will never go away and thus be retrievable in the future. Of course, this is an illusion, and we will see how this illusion is achieved, but it is a key property

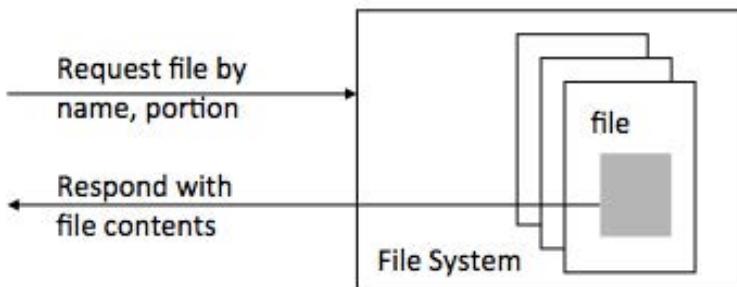


Figure 11.1: A file system supports storage and retrieval of files.

that users expect from a file system.

The notion of a file in a file system is very general, thus sometimes referred to as an “object.” These objects may be data or programs, and they may be for the system or for users. These objects are referenced by name, to be read or written. The file system abstraction gives the illusion of persistence, such that these objects will exist and remain accessible without time limit, essentially “forever.” It provides the illusion of unbounded storage space. It provides access control, allowing for degrees of sharing between users. And it provides security, maintaining privacy, integrity, and accessibility of the objects it contains.

11.1 File System Characteristics

In fact, the file system is more than a repository in that it can contain and manage any and all objects that have the following characteristics:

- accessible by name

- can be read and/or written
- can be protected by specifying permissions, e.g., read, write, read-only, etc.
- can be shared
- can be locked

While these characteristics naturally apply to objects such as data and programs, they can also be applied to I/O devices, such as a disk, a keyboard, a display, etc. Indeed, they can also be applied to the memory of processes. While the term object captures the generality of including data, programs, devices, etc., we will use the term file, now recognizing that it is a highly general concept.

The file system is generally the most visible part of the operating system for most users, and so its user interface is especially important. A key part is the file name space, which is typically hierarchical. By this we mean that the name space is organized as a tree. A file name has components, each component corresponding to a branch of the tree, starting from a root. Organizing files in a hierarchy, and having this hierarchy expressed in the structure of names, is both convenient and intuitive for users.

11.2 File Name Space

A common example of a hierarchical file name space is that used by the UNIX operating system; in fact, most operating systems use similar ideas. (We will be using the UNIX file system as a model throughout this chapter, given its popularity and influence. Understanding the UNIX file system provides a good base of knowledge for both understanding how file systems work in general, and for learning how file systems of other operating systems work.) In UNIX, a file name is called a “path name” because it describes a path through the file system’s file name hierarchy, or tree. Figure 11.2 shows a simple example of such a tree.

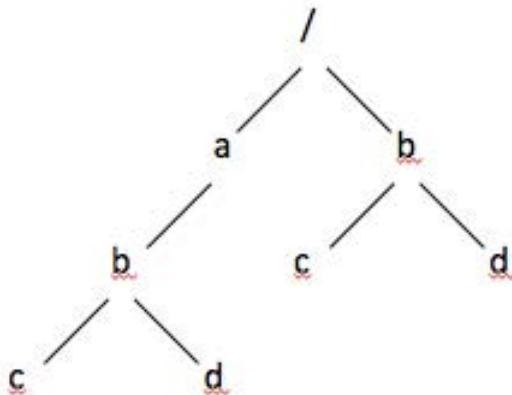


Figure 11.2: A file name hierarchy or tree.

An *absolute* path name describes a path that always begins at the root of the tree, indicated by the “/” (slash) character as the leading character of the name. For example, “/a/b/c” is the name of the file located at the lower

left part of the tree.

A *relative* path name describes a path that is relative to some other name; it is distinguished from an absolute path name because it will not have a leading “/”. For example, the “b/c” is a relative path name. Looking at the tree in Figure 11.1, there are two possibilities for which file “b/c” refers to. Relative to “/a”, “b/c” refers to the file located at the lower left part of the tree. Relative to “/”, “b/c” refers to the file located at the lower middle part of the tree.

In fact, the UNIX file system name space is not strictly hierarchical, as a file can actually have multiple names, called links.

11.3 File Attributes and Operations

A file has *attributes*, or characteristics that have meaning to the system.

Common attributes include:

- a type
- times, such as creation, last access, last modified
- sizes, such as the current size and maximum size
- access control permissions, such as read, write, execute

A file can be manipulated via a set of *operations*. Common operations include:

- creation: create, delete
- preparation (for access): open, close, mmap
- access: read, write
- search: move to location
- attributes: get, set (e.g., permissions)
- mutual exclusion: lock, unlock
- name management: rename

These operations are expressed in a program using *system calls*, which form the interface the kernel provides to user processes. Each system call is recognized by the compiler, and is translated into a sequence of instructions that place parameters in registers (typically) and issuing the TRAP instruction, which causes control to go to the kernel. The TRAP instruction will typically have an operand, expressed as TRAP x , where x corresponds to the particular system call being invoked. This allows the kernel to jump to the corresponding section of its code that implements that particular system call.

11.4 The Read/Write Model

There are two common models for file access, the read/write model, and the memory-mapped model. A common system call interface (in fact, the one defined by UNIX) for the *read/write model* is shown in Figure 11.3.

```

fd = open (filename , mode)
nr = read (fd , buf , n)
nw = write (fd , buf , n)
close (fd)

```

Figure 11.3: System calls for the read/write model of file access.

To access a file, it is first opened, via the `open` system call, after which it can be read or written, via the `read` and `write` system calls, as many times are needed, and when done, is closed, via the `close` system call. Their use is illustrated in Figure 11.4.

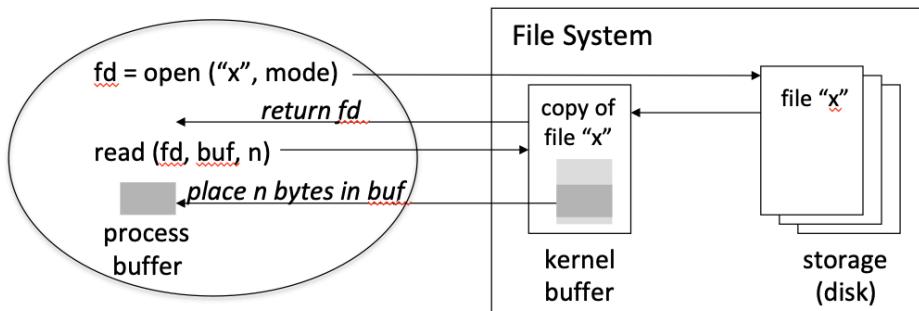


Figure 11.4: The read/write model of file access.

The `open` system call informs the kernel that the file is to be accessed (with subsequent reads and/or writes), allowing it to perform checks and make preparations for file access. By providing the file name, the kernel will find the file's control information, called *metadata*, which will include its attributes such as permissions as well as where to find its contents. This allows it to check the permissions to see whether the process is allowed to

access the file, and if so, in what way, e.g., read-write, read-only, etc.

Note that the file – both its metadata and contents – will be permanently stored on a disk, which is relatively very slow compared to physical memory. Consequently, as part of the preparations for accessing a file, the kernel may even read a portion of the file into physical memory, in the kernel’s data area. This “pre-staging” places the data in a more favorable position, so that if and when the process issues read or write system calls, the contents will already be in the physical memory, rather than having to wait for the disk access.

This temporary holding area in memory is called a *buffer*. The buffer is in the kernel (in its data area), consequently it is a *kernel buffer*. It is in the kernel’s rather than the process’s memory since the process has yet to actually request the data. Furthermore, if some other process were to open the same file, there would be no need to do another pre-staging in the kernel’s memory because it is already there.

Assuming the open system call is successful, it will return a *file descriptor*. This is a temporary identifier that the kernel recognizes (because it created it) as corresponding to this particular open call, and is in effect until the close system call is made, signifying the end of use of the file. Calls to `read` and `write` (and eventually `close`) will take the file descriptor as a parameter, rather than the file name. We will discuss why in more detail below, but suffice it to say that there is significant work involved in processing the file name to determine the location of the file’s metadata and contents.

Once found, to avoid repeating this work, the kernel stores the resulting

location information, which is referred to by the file descriptor. Thus, the file descriptor is fast index to the open file; when the process presents the file descriptor to the kernel, the kernel already knows how to quickly locate the file metadata and contents, whereas finding them using the file name takes much longer.

The process can then read a portion of the file's data by issuing the `read` system call, which takes a file descriptor (`fd`) as a parameter that identifies the file. It also takes a memory address (`buf`) in the process's memory indicating where the data should be placed when it is read, and an integer (`n`) that specifies how many bytes should be read.

An implicit *open file pointer* (a per-open file variable maintained in the kernel), initialized to the beginning of the file, determines the portion of the file read. Consequently, if a file is opened and a `read` system call is issued to read n bytes, it is the first n bytes of the file that are read. The open file pointer is then updated, pointing to the byte after what was just read. Thus, subsequent `reads` (and `writes`) will read (and write) relative to where the pointer last left off.

The read/write model is a *buffered* model of file access, in that the process receives a *copy* of the file's data, which it can then process in any way it wishes (it can be used in computations, it can be modified, etc.). Since this is a copy, the actual data of the file is not affected. This is why this receiving area in the process's memory is called a *buffer*, as it is a temporary holding area that is distinct from where the data came from.

In saying that the read/write model is a “buffered” model of file access, this is from the point of view of the user/programmer. The programmer must explicitly take note that the process, when operating on file data in its memory obtained from a `read` system call, is not operating directly on the file (which is located on disk), but rather a copy, and that until a `write` system call is made, the file has not been updated.

What about the buffering that occurs in the kernel, as discussed earlier? Unlike the buffering we just discussed that happens in the process’s memory, the buffering in the kernel is *invisible* to the process; in fact, its main purpose is to improve performance. We will discuss this in detail the next chapter on I/O (and in fact, it is called a buffered model of *I/O*, distinguished from our usage here of buffered model of *file access*). So, when we say that file access is based on a buffered model, we strictly mean the model as viewed by the user/programmer, which has logical consequences (i.e., the programmer must be aware of this in designing a program), and not whether the kernel itself is doing buffering.

The `read` system call, if successful, will return the actual number of bytes read, which may or may not match the number of bytes that were requested to be read. It should match, unless what was requested is beyond what remains in the file. For example, if there is a request to read 1000 bytes, and there are only 500 bytes in the file, there is a mismatch and what will be returned is 500, indicating what was actually read than what was requested.

If the file is to be updated, the process will issue the `write` system call,

which, like `read`, takes a file descriptor that identifies the file. Also, like `read`, it takes a memory address (`buf`) and integer (`n`) that indicate where to find the data (in the process's memory) to be written to the file and how many bytes. This then overwrites the portion of the file starting pointed to by the open file pointer.

To clarify how the open file pointer works, consider the following example. Say a file is opened, and a `read` system call is issued for 1000 bytes. This will read the first 1000 bytes into the process's memory. The open file pointer, which was initialized to the first byte after the file was opened, points to the 1001st byte after the read. If another `read` is issued for 1000 bytes, this will read the second 1000 bytes into the process's memory, and the open file pointer will point to the 2001st byte. Say that the process modifies the 2000 bytes of file data that was copied into its memory, and then writes all 2000 bytes. This will *not* update the first 2000 bytes of the actual file, because the open file pointer was pointing to the 2001st byte; consequently, the second 2000 bytes are updated (and the open file pointer will then point to the 4001st byte). This may not be what the programmer desired, but it is how it works.

If the desire was to update the first 2000 bytes of the actual file, the open file pointer must somehow be reset to point to the first byte. This can be done in either of two ways. First, the file can be closed (by calling the `close` system call), and then opened again, getting a new file descriptor, and the open file descriptor associated with this file descriptor will be initialized to the first byte. Issuing the `write` system call of the 2000 bytes in the process's

memory, but with this file descriptor, will update the first 2000 bytes of the file, as desired. In fact, there is actually no need to have closed the file, as the first `open` call and the second are separate, and each have their own open file pointers, updated separately based on which of their respective file descriptors are used.

The second way of updating the first 2000 bytes is to reset the open file pointer, which can be done via a separate system call we did not mention above: `lseek (fd, offset, whence)`. The first parameter is the file descriptor, the second is an offset relative to some reference point, and the third parameter selects the reference point, which can be the beginning of the file, the current position of the open file pointer, or the end of the file, specified by the values 0, 1, or 2, respectively. Thus, `lseek (fd, 0, 0)` will set the open file pointer to the beginning of the file. The write system call can then be made, and this will overwrite the first 2000 bytes in the file. This second method does not require opening the file again, and so the same file descriptor would be used for all the system calls.

When a `write` system call is made, we said that the “actual file is updated.” While this is *logically* true (i.e., as reasoned by the programmer, the file was updated), this may not be true in reality, at least for temporarily. Specifically, recall that the kernel will itself buffer file data. When data is written to a file, it may initially be stored in a kernel buffer, and eventually written out to disk rather than immediately. This is done again for performance reasons. If the file is updated, and updated again very soon, there

can be a savings of a disk access if only the final update is actually written to disk.

Another reason is that the amount of data that can be written to a disk may be limited to a certain block size, such as 1024 bytes. If the process writes out the first 100 bytes, the kernel will have had to have the entire block of the first 1024 in a kernel buffer, modify the first 100 bytes, and then write out the entire block to disk. If it delays this write, and a second write system call is made of any of the data in that first block, the kernel buffer containing that block will be updated, and then the entire block can be written to disk, having saved the first disk write request.

There is a tradeoff between performance and reliability here: the more delay introduced, the more efficient the disk will be used as the number of disk writes will be reduced. But, if the system crashes, and blocks that were buffered in the kernel and have not been written to the disk are then lost. Most operating systems will set a limit on the maximum amount of time a block can be buffered in the kernel before it is written to disk. For example, if this time were set to 30 seconds (a typical value), then a user would at worst lose the last 30 seconds of a document they were modifying. This may be acceptable to the user if the alternative was much slower general system performance due to more frequent updating of the disk.

11.5 The Memory-Mapped Model

The second common model of file access is the *memory-mapped model*. In this model, a file, either a portion or its entirety, is *mapped* into the process's virtual memory address space, and then when that part of the memory is read or written, the underlying file is correspondingly read or written.

A simplified system call interface for the memory-mapped model is shown in Figure 11.5.

```
fd = mmap (filename , addr , offset , n)
close (fd)
```

Figure 11.5: System call for the read/write model of file access.

The `mmap` system call maps an *n*-byte portion of the file given by file name, starting at `offset` into the process's virtual address space at address `addr`. The act of “mapping” causes the bytes from `offset` to `offset+n-1` of the file to be accessible by the process by reading and writing the memory addresses `addr` to `addr+n-1`.

As an example of its use, say there is a variable `xptr` declared as a pointer, and then set to `addr`: `xptr = addr`. Then, after having called `mmap (fd, addr, offset, n)`, the statement `x = *xptr` will read the byte at `offset` bytes from the beginning of the file into the variable `x`, and `*xptr = 1` will write that same byte of the file with the value 1. Setting `xptr = addr + 6` and then doing the same read and write operations (`x = *xptr` and `*xptr`

- = 1) will read the byte at `offset+6` and write a 1 into the byte at `offset+6`, respectively, of the file.

The interface allows a file to be mapped in its entirety, or just a small portion, and that mapping can be located anywhere in the process's virtual address space. Figure 11.6 illustrates two files that are memory mapped into the virtual address space, the small one in its entirety, and the larger one, only a portion.

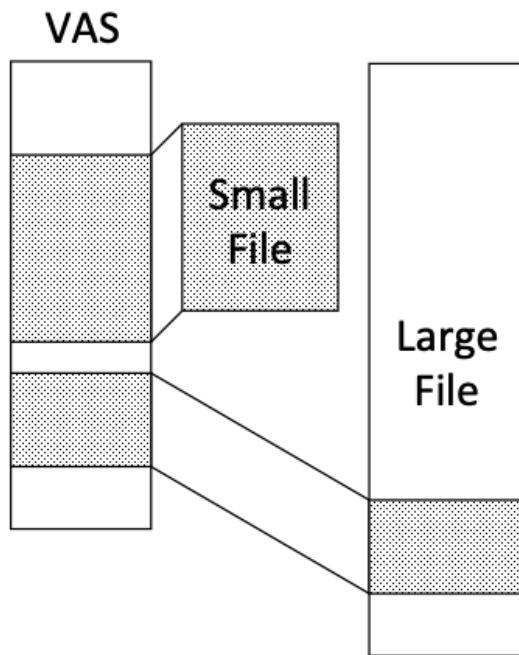


Figure 11.6: Two memory mapped files.

To the process, the reading and writing are operating on the actual file. Consequently, this is *not* a buffered model of file access, where the process is reading and writing into memory containing a copy of the file data. This is

a critical difference between the memory-mapped and read/write models.

However, the way this is *implemented* will involve buffering, but the buffering will occur in the kernel. This must be the case, as a secondary storage device, such as a disk, will not permit the reading or writing of a single byte, or any amount that does not correspond to the device's block size (typically 512, or 1024, or some multiple of 1024 bytes). Thus, an update to a particular byte must involve reading the block containing that byte into the kernel's memory, then reading or writing that byte, and in the case of a write, then writing the modified block out to disk. The copy of the block in the kernel memory corresponds to a buffer. But, as this is not visible to the process, and thus the programmer, the memory-mapped model is not a buffered model of file access.

11.6 Comparison of the Models

Both of the models have their advantages and disadvantages, mostly having to do with what the programmer finds most natural to use. The memory-mapped model allows direct reading and writing of a file's contents, once the file is mapped into the process's memory address space. There is no opening, reading, and writing of based on the use of a copy of the file's contents. If two processes memory map the same file, then updating a byte corresponding to the file by one process will be immediately visible to the other process.

This is in contrast to the read/write model, where if two processes open

the same file, the processes operate on their own separate copies of the file data until a write system call operation is made, after which a read system call by the other process must be made to see the effects of the update. If the programmer finds it more natural for the process to work on a copy of the file data until it is explicitly written, then the read/write model would be preferred over the memory-mapped model.

There are significant performance implications of the two models. Say there are two processes, both of which are to operate on the same file at the same time. With the memory-mapped model, there is an expectation that an update by one process, by simply having assigned a value to a variable, will be immediately visible to the other process via its memory. Having this occur instantaneously (i.e., an update by one process is immediately visible by the other process) may be difficult to support, as would especially be the case if each of the processes is running on a separate CPU in a multiprocessor machine. This is much less of an issue with the read/write model, as updates only become visible after an explicit call to the write system call, followed by an explicit call to the read system call. System calls are much heavier-weight operations, and occur at a lower rate, than assigning and updating of variables (i.e., read and write system calls will be spaced out in time much more than variable assignments).

11.7 Access Control

An important expectation by users is that they be able to *share* their files, with specific users and to varying degrees for each. A user may wish that a particular file be shared with certain specific colleagues, and that the file be modifiable by a subset while the others have only read access. The user may wish that other files be completely private, inaccessible to any other user, while some files should be completely public, accessible to any users without specification (and thus even users not known or existing at the time the file was created).

Access control is what provides these various levels of sharing (along with the ability to express them), and requires a *mechanism* that is part of the file system to support it. Thus, access control supports, on a per file basis, which user or users can access a file, and what operations are allowed for that user or group. The specific permissions that are allowed for a specific file (or group of files) is the access control *policy* for that file or file group.

The file system must also provide a user interface that allows the owner of a file to be able to specify the access control policy for that file. This user interface must be simple and intuitive, as if it is too cumbersome to use, users will either not use it and thus rely on default access control settings, or may use it to express what is easiest to specify, which may be different than what they actually desire.

Access control in the UNIX operating system's file system is a good ex-

ample of a well-designed interface and mechanism. Access control of a file is settable by the file's owner, and is expressed as a set of permissions for the file's owner, for a specific group of users, or for everyone (all users, known and unknown to the owner). The specific group of users is specified independently, and is given a name so that the group can be referred to by name in access control permissions for a specific file

The allowable permissions include the ability to read the file, to write/- modify the file, and to execute the file, indicated in the user interface by the letters “r”, “w”, and “x”. This access control specification is encoded in a 9-bit word, partitioned into 3 groups of 3 bits, where the presence of the permission is indicated by the letter, and lack of that permission is indicated by a dash “-”. The first 3 bits indicate whether read, write, and execute access is allowed for the file's owner, the next 3 bits for the specific group of users, and the last 3 bits for the group containing everyone.

For example, the bit string “rw-r—” indicates that the owner can read and write the file, the group of users associated with the file (specified separately) can read the file, and everyone else has no permission. The bit string “rwxr-xr-” indicates that the owner can read, write, and execute the file, the group of users associated with the file can read and execute (but not write) the file, and everyone else can only read the file.

We say that this is a well-designed interface because it is fairly easy to understand, is concise enough that it doesn't take much effort to express what the file's owner desires, and captures most of what users will generally wish

to express. If a user wishes to express an access control policy that is beyond this simple form of specification (for example, that a group of users be able to only append to the end of the file rather than the more general “write” permission, which allows modifying of what already exists in addition to appending), UNIX provides a separate mechanism, called “SETUID,” which will be discussed in the chapter on Protection. The SETUID mechanism is more general, and correspondingly significantly more complex, than the 9-bit specification described above. This combination of mechanisms capture an important design principle of UNIX, where the common case is handled by a simple and easy-to-understand mechanism, but that uncommon cases are still supported, albeit with a separate complex mechanism (that most users need not be bothered with learning if they have no need for dealing with the uncommon cases).

11.8 File System Implementation

The abstractions of files and the file system must ultimately be realized in mechanisms and policies that rely on underlying storage resources. These storage resources depend on technologies that are numerous, come in different varieties, and have very different characteristics. The two most common storage technologies are hard disk drives (HDD), typically rotating magnetic disks, solid state drives (SSD), typically non-volatile flash memory. HDDs are less expensive than SSDs, thus providing larger total storage per dollar,

while SSDs are faster, more reliable, and consume less power. A file system may be implemented using a combination of these technologies (as well as others, and including storage that is remotely accessed via a network), in a way that hides these differences. The overall goal is to present the illusion of a large repository of storage whose contents remain intact (independent of the availability of electrical power) and available, forever.

11.9 Goals

This last point can be expanded and elaborated into a number of goals that we seek to achieve in implementing a file system. These include:

- Persistence: once files are saved, they should remain forever
- Speed: contents should be stored and retrieved without perceptible delay
- Integrity: the contents must remain as originally saved, allowing only for changes that are intended (i.e., bits should not change in some arbitrary or unauthorized way over time)
- Availability: when a user wishes to retrieve a file, they should be able to do so immediately
- Archival: if a file is modified, the previous contents should remain available so that a user can go back to previous versions

- Technological variety: the actual storage can be built from any of a number of storage technologies that can have different characteristics (in terms of speed, reliability, cost, etc.)
- Scalable: the amount of storage can be incrementally increased over time, depending on the need for extra storage
- Local and remote: the storage may be local in that it is directly connected to the computer system, or remote such that it is accessible over a network
- Privacy: contents should not be available to anyone (including the system operator) other than those specifically authorized by the owner of the contents
- Cost: keeping costs low is always a goal, but is especially important in file system implementation given the exponential growth of data and users' insatiable desire for storing everything

The above goals may be further categorized as aspects of the higher goals of reliability, performance, and security. The art of implementation will then depend on how to best achieve these goals, and will involve balancing trade-offs in what are competing goals. For example, security may be improved by encrypting all stored information, but this can decrease performance as encryption and decryption add delay to storage and retrieval. Reliability may be improved by storing multiple copies, but this can decrease performance

as managing multiple copies increases overhead. If the copies are stored in remote places, security may also be decreased because the information may be more widely accessible. Finally, an increase in performance, reliability, or security, will cause an increase in cost, for which there is always a limit on how much is acceptable.

Ultimately, each installation will have to decide how to emphasize the goals relative to each other. A flexible implementation will allow some parameterization so that the relative importance of these goals can be dynamically changed.

11.10 Abstract Storage Device

Since the implementation of a file system is dependent on the use of some particular hardware storage device, a complete description would have to include what specific operations are issued to that device, and when. Given that there are many different kinds of devices, any description would become overly complicated by discussing each of the devices' specific operations, each of which may be different. This would distract from our goal, which is to describe how the file system is implemented, independent of the underlying storage device being used. And yet, the underlying storage device is a key part of the file system's operation, and so it must enter into the discussion.

We resolve this issue by abstracting the storage device's operation to that of an “abstract storage device,” keeping only those features of storage

devices that help explain the implementation of the file system, and not dwelling on device-specific issues. This is not to say that those device-specific features are unimportant; in fact, we will address some of those features in the chapter on I/O when we discuss device drivers. For now, all we need is a general interface to the abstract storage device, allowing us to discuss the file system's implementation in terms of the use of that interface, so let us deal with this now.

Our abstract storage device is modeled as an array of N fixed-size blocks, indexed and randomly addressable by block number 0 through $N - 1$, as shown in Figure 11.7, where $N =$ the storage device's capacity divided by the block size. A typical block size is 1KB (which we will use in our examples), but larger sizes that are multiples of 1KB, such as between 4KB and 64KB are also common.

We use fixed-size blocks because storage devices, in general, allocate and transfer data in fixed-size blocks. A typical minimum block size for a disk drive is 512 bytes. If our file system design assumes that the block size of the abstract storage device is 1KB, this would simply mean that blocks on the actual hard disk drive would be accessed in consecutive pairs of 512-byte blocks so that they effectively are allocated and transferred as 1 KB blocks.

The software interface of the abstract storage device comprises two functions, `read_block` and `write_block`, as shown in Figure 11.8. The function `read_block` transfers the block addressed by `block_num` from the storage device into a block of physical memory starting at the address `mem_addr`.

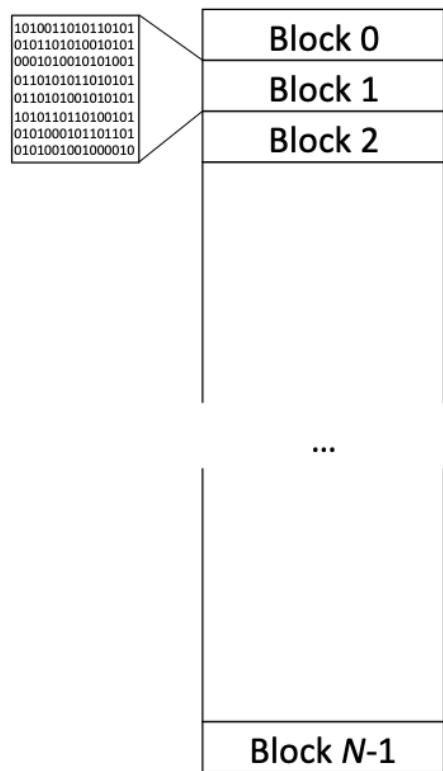


Figure 11.7: Storage modeled as an array of N fixed size blocks, numbered 0 to $N - 1$.

The function `write_block` transfers a block of data from physical memory at the address `mem_addr` to the block addressed by `block_num` on the abstract storage device. Our file system implementation, when it needs to access the abstract storage device, will do so solely in terms of these two functions.

```
read_block (block_num, mem_addr)  
write_block (block_num, mem_addr)
```

Figure 11.8: Software interface of abstract storage device.

A simple structure for the abstract storage device, for the purposes of storing the entire file system, is shown in Figure 11.9. The storage is partitioned into three regions of consecutive blocks:

- Region 1: File System Metadata – information about the entire file system
- Region 2: File Metadata – information about each file
- Region 3: Data Blocks – data that comprises each file

The first region, *File System Metadata* (FSM), contains information about the entire file system, and is typically just a few blocks in size. The second region, File Metadata (FM), contains file control blocks, one per file, each of which contains information about a particular allocated file. The third region, Data Blocks (DB), by far the largest region, contains the actual contents of files; consequently, a file will be allocated a set of data blocks that contain the data for that file.

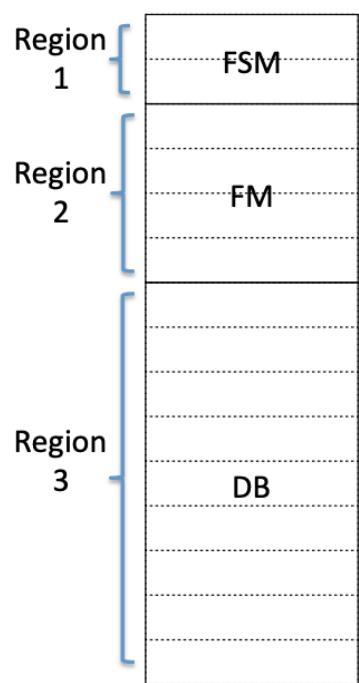


Figure 11.9: Simple file system implementation structure on storage device.

Creating and initializing this structure is commonly referred to as “formatting” the file system. Before proceeding, we note that this is only one way of organizing how a file system is structured onto a storage device; there can be and certainly are other ways. We chose this as it is derived from a real file system (an early version of UNIX), and it is fairly simple to understand. It does have its limitations; for example, each region is assumed to have a certain number of blocks (the sizes specified as part of the formatting procedure) that will not change, so if it is later determined that, say, not enough space was allocated to the FM region, the only way to fix this is to reformat the storage device (first copying the file system to some other place so that it is not lost and can then be copied back to the newly formatted device). Ways of structuring the storage device have improved over the years, doing away with some of these limitations, but for our purposes, the simple structure we describe will suffice. If you understand this structure, you will be able to understand more complex organization schemes.

The FSM contains information that includes global parameters, such as what block size is being used throughout the storage device and thus for the file system, how big the file system is (in terms of the specified block size), what are the starting block locations for the file metadata region and the data block region so the file system knows where to find these regions, how many files are allocated (i.e., how many file control blocks, discussed below, are allocated), how many data blocks are in use, etc. The FSM will also include free lists that indicate which file control blocks are free, and which

data blocks are free. Since the FSM is assumed to be located at the beginning of the storage device, there is no need to record where it is, as it is known by default.

The second region, *File Metadata* (FM), contains information about individual files. This region is structured as an array of *file control blocks*, one per file, as shown in Figure 11.10. Each file control block is indexed by its position in the array (starting at 0), and this index is called the file control block number. Note that while the File Metadata region is composed of a sequence of storage blocks, a storage block and a file control block are not the same. Typically, a file control block is much smaller than a storage block, and so many file control blocks will fit in a single storage block. A better way to view this is that however many storage blocks comprise the File Metadata region, their combined space is used to store the file control block array, which is essentially superimposed on this region.

A file control block contains all the information needed to access a file. This includes attributes, such as the type of file (regular or system, and possibly other categories), the size of the file, permissions, etc.

Importantly, the file control block also contains a *disk block map*, which specifies which blocks contain the actual data of the file. For example, say that the file is of size 4000 bytes (numbered 0-3999). Assuming a storage block size of 1024, to store the file's data would require 4 blocks, providing 4096 bytes of storage (the residual 96 bytes, considered internal fragmentation but now applied to disk storage rather than physical memory as discussed

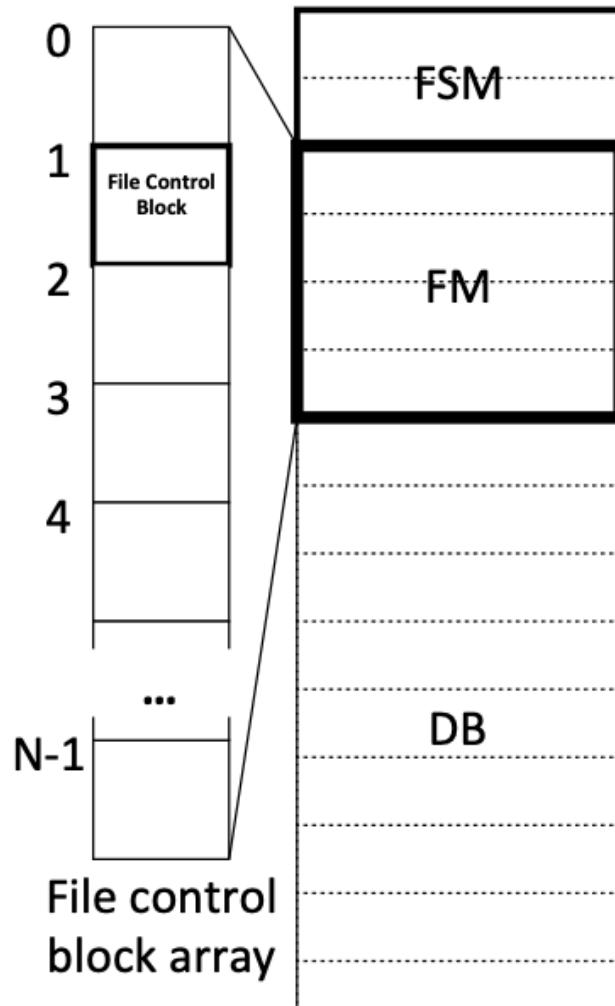


Figure 11.10: The File Metadata region structured as an array of file control blocks.

in Chapter 8, would be unused but available if the file were to grow).

We would still need to know *which* of all the storage blocks are these four, and so their block numbers (indexes into the array used to model the storage device) would be specified in the disk block map. If the block numbers are 7076, 9201, 567, and 9248, then block number 7076 (i.e., the storage block indexed by 7076) would contain bytes 0-1023 (i.e., the first 1024 bytes), block number 9201 would contain bytes 1024-2047, block number 567 would contain bytes 2048-3071, and block number 9248 would contain the remaining bytes 3072-3999 of the file, along with bytes 4000-4095 not being used.

There are a number of data structures that can be used to realize the disk block map. The disk block map essentially translates block numbers, which are logical values (they determine which block in the block array that comprises the abstract storage device

11.11 Summary

In this chapter, we learned that a ... However, we have not discussed any policy. That will have to wait for the chapter after next.

11.12 Exercises

1. What is a file?
2. What is a file system?

3. Why is the word “system” used in “file system”; why not simply call it a “file collection”?***
4. By “repository of objects”, what is meant by “repository” and what is meant by “object”; why not say a “repository of files”?**
5. What are the features of an object as it applies to files?*
6. For each of the following I/O devices, how can it be viewed as a file object: disk, keyboard, display, mouse, printer?**
7. How can the memory of a process be viewed as a file?**
8. What is a name space?*
9. What makes a name space hierarchical?**
10. What is the value of a hierarchical name space?*
11. Why are UNIX files names called “pathnames”?*
12. What’s the difference between absolute and relative path names?*
13. What is meant by the word “attribute”?*
14. What criteria would be used to determine that an attribute qualifies as a “system attribute”?***
15. For each of the following, justify why they are system attributes: type (system or user), creation time, access time, modified time, current size, maximum size, permissions?***

16. What are the operations allowable on files?
17. Some operations are issued via system calls, and some are not: for each of the operations in the previous question, argue why it should or should not be issued via a system call?***
18. If an operation is deemed not to require a system call, then how would the operation be issued?**
19. What is the buffered Read/Write Model for file access?*
20. What are the system calls for this model?*
21. For each of the system calls, why are they necessary, or why are they not necessary?***
22. For each of the system calls, why are each of the parameters and return value needed: justify each one?**
23. More generally, what criteria would determine whether a procedure call needs to be a system call?***
24. The Standard I/O Library has a file interface that includes fopen, fread, fwrite, and fclose: how are these different from open, read, write, and close, and why are they not system calls?**
25. What is the Memory-Mapped Model for files?

26. What parameters are needed for the mmap system call, and in what way are they used?**
27. What are the pros/cons of mmap?**
28. How can mmap be used for processes that share memory?**
29. What complication arises if an mmapped file is modified?** What are solutions for this?***
30. What is access control?
31. In what way does access control support the sharing of files?*
32. What is the access control interface for UNIX?**
33. How is access control in UNIX different from that of MULTICS?***
34. What are the goals for a file system implementation?
35. Why must the following be “balanced”, as opposed to “maximized”: performance, reliability, security?**
36. What is the storage abstraction, and what is its value?*
37. How is the storage abstraction interface used (explain the functions and parameters)?**
38. In the typical implementation structure, what are the three regions, justify why each is necessary and sufficient?***

39. What is file system metadata?
40. What kind of information comprises file system metadata, and what are examples?*
41. For each of the examples you present, can you justify why they are file system metadata (as opposed to the other categories of file system information)?**
42. What is a free list?*
43. What is a bit map?*
44. For each, when is it better to use one vs. the other?**
45. What is file metadata?
46. What kind of information comprises file metadata, and what are examples?*
47. For each of the examples you present, can you justify why they are file metadata (as opposed to the other categories of file system information)?**
48. What is a file control block?*
49. For the example of a file control block on Figure 11.16, what is each one and why is it necessary?*

50. What are the pros and cons of having the file's name be located in the file control block?**
51. What are the 3 basic ways for keeping track of allocated blocks; explain how each works?**
52. In Figure 11.17, two methods are shown for the non-contiguous block method: how does each work, and when would one be used vs. the other?**
53. What is the UNIX Version 7 Block Map, and how does it work?**
54. What are indirect pointers, and what is the purpose of having them?**
55. Can you explain the diagram in Figure 11.19, and how the maximum file size is calculated?**
56. If the block size is doubled, how does this affect the maximum block size?***
57. What are the three methods of keeping track of free blocks; how does each work?*
58. What criteria would you use for deciding which method to use?**
59. Why is it necessary to convert file names to file control blocks?**
60. How is this conversion done in pre-BSD UNIX (and by example, can you explain the diagram in Figure 11.23 that shows how the UNIX file name “/sports/baseball/Padres” is converted)?**

61. In “The Big Picture” diagram in Figure 11.24, how many blocks need to be accessed to get the data contents of the file “dog”, assuming the first required access is to the File System Metadata (and explain each of the accesses)?***
62. How does a mechanical magnetic disk work, based on the diagram in Figure 11.25?*
63. Why are mechanical magnetic disks used for file system storage?*
64. What is the biggest negative in using a magnetic disk, as it relates to system performance?**
65. What is the average time for a disk access?*
66. What are the components of this time?
67. For each component, can you think of a way of reducing its time, and if so, by how much do you think it can be reduced?***
68. What is the main idea (can be stated in 3 words) for dealing with the time expense of using mechanical disks?**
69. What is a Solid State Drive (SSD)?
70. In what ways are SSDs different from mechanical hard disk drives (HDD)?*
71. Why does an SSD wear out with age?**

72. What is the biggest advantage SSDs have over HDDs?*
73. What is the biggest disadvantage SSDs have over HDDs?*
74. How can caches be used to improve file system performance?*
75. What kinds of information can be cached?**
76. For each kind of information, are there any special concerns as to caching that kind relative to the others?**
77. What is clustering, and how can it be used to improve file system performance?*
78. Why does clustering improve performance?*
79. How is block size related to performance?
80. Explain the effect on performance by making the block larger? Why this effect?*
81. Explain the effect on performance by making the block smaller? Why this effect?*
82. Given these effects, should the block size be made smaller or larger?*
What goes into making this decision?**
83. What is meant by file system reliability?*
84. What is meant by consistency?*

85. How does consistency affect reliability?**
86. How can consistency be increased?*
87. Are there drawbacks to increasing consistency, and if so, what are they?*
88. What is meant by journaling?*
89. How does journaling affect reliability?**
90. What are the basic steps to journaling?*
91. How is a journal used after a crash?**