Preface

To Everyone

Welcome to this book! We hope you'll enjoy reading it as much as we enjoyed writing it. The book is called **Operating Systems: Three Easy Pieces**, and the title is obviously an homage to one of the greatest sets of lecture notes ever created, by one Richard Feynman on the topic of Physics [F96]. While this book will undoubtedly fall short of the high standard set by that famous physicist, perhaps it will be good enough for you in your quest to understand what operating systems (and more generally, systems) are all about.

The three easy pieces refer to the three major thematic elements the book is organized around: **virtualization**, **concurrency**, and **persistence**. In discussing these concepts, we'll end up discussing most of the important things an operating system does; hopefully, you'll also have some fun along the way. Learning new things is fun, right? At least, it should be.

Each major concept is divided into a set of chapters, most of which present a particular problem and then show how to solve it. The chapters are short, and try (as best as possible) to reference the source material where the ideas really came from. One of our goals in writing this book is to make the paths of history as clear as possible, as we think that helps a student understand what is, what was, and what will be more clearly. In this case, seeing how the sausage was made is nearly as important as understanding what the sausage is good for[1].

There are a couple devices we use throughout the book which are probably worth introducing here. The first is the **crux** of the problem. Anytime we are trying to solve a problem, we first try to state what the most important issue is; such a **crux of the problem** is explicitly called out in the text, and hopefully solved via the techniques, algorithms, and ideas presented in the rest of the text.

In many places, we'll explain how a system works by showing its behavior over time. These **timelines** are at the essence of understanding; if you know what happens, for example, when a process page faults, you are on your way to truly understanding how virtual memory operates. If you comprehend what takes place when a journaling file system writes a block to disk, you have taken the first steps towards mastery of storage systems.

There are also numerous **asides** and **tips** throughout the text, adding a little color to the mainline presentation. Asides tend to discuss something relevant (but perhaps not essential) to the main text; tips tend to be general lessons that can be

[1] Hint: eating! Or if you're a vegetarian, running away from.

applied to systems you build. An index at the end of the book lists all of these tips and asides (as well as cruces, the odd plural of crux) for your convenience.

We use one of the oldest didactic methods, the **dialogue**, throughout the book, as a way of presenting some of the material in a different light. These are used to introduce the major thematic concepts (in a peachy way, as we will see), as well as to review material every now and then. They are also a chance to write in a more humorous style. Whether you find them useful, or humorous, well, that's another matter entirely.

At the beginning of each major section, we'll first present an **abstraction** that an operating system provides, and then work in subsequent chapters on the mechanisms, policies, and other support needed to provide the abstraction. Abstractions are fundamental to all aspects of Computer Science, so it is perhaps no surprise that they are also essential in operating systems.

Throughout the chapters, we try to use **real code** (not **pseudocode**) where possible, so for virtually all examples, you should be able to type them up yourself and run them. Running real code on real systems is the best way to learn about operating systems, so we encourage you to do so when you can.

In various parts of the text, we have sprinkled in a few **homeworks** to ensure that you are understanding what is going on. Many of these homeworks are little **simulations** of pieces of the operating system; you should download the homeworks, and run them to quiz yourself. The homework simulators have the following feature: by giving them a different random seed, you can generate a virtually infinite set of problems; the simulators can also be told to solve the problems for you. Thus, you can test and re-test yourself until you have achieved a good level of understanding.

The most important addendum to this book is a set of **projects** in which you learn about how real systems work by designing, implementing, and testing your own code. All projects (as well as the code examples, mentioned above) are in the **C programming language** [KR88]; C is a simple and powerful language that underlies most operating systems, and thus worth adding to your tool-chest of languages. Two types of projects are available (see the online appendix for ideas). The first are **systems programming** projects; these projects are great for those who are new to C and UNIX and want to learn how to do low-level C programming. The second type are based on a real operating system kernel developed at MIT called xv6 [CK+08]; these projects are great for students that already have some C and want to get their hands dirty inside the OS. At Wisconsin, we've run the course in three different ways: either all systems programming, all xv6 programming, or a mix of both.

To Educators

If you are an instructor or professor who wishes to use this book, please feel free to do so. As you may have noticed, they are free and available on-line from the following web page:

`http://www.ostep.org`

You can also purchase a printed copy from `lulu.com`. Look for it on the web page above.

The (current) proper citation for the book is as follows:

Operating Systems: Three Easy Pieces
Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau
Arpaci-Dusseau Books
March, 2015 (Version 0.90)
`http://www.ostep.org`

The course divides fairly well across a 15-week semester, in which you can cover most of the topics within at a reasonable level of depth. Cramming the course into a 10-week quarter probably requires dropping some detail from each of the pieces. There are also a few chapters on virtual machine monitors, which we usually squeeze in sometime during the semester, either right at end of the large section on virtualization, or near the end as an aside.

One slightly unusual aspect of the book is that concurrency, a topic at the front of many OS books, is pushed off herein until the student has built an understanding of virtualization of the CPU and of memory. In our experience in teaching this course for nearly 15 years, students have a hard time understanding how the concurrency problem arises, or why they are trying to solve it, if they don't yet understand what an address space is, what a process is, or why context switches can occur at arbitrary points in time. Once they do understand these concepts, however, introducing the notion of threads and the problems that arise due to them becomes rather easy, or at least, easier.

As much as is possible, we use a chalkboard (or whiteboard) to deliver a lecture. On these more conceptual days, we come to class with a few major ideas and examples in mind and use the board to present them. Handouts are useful to give the students concrete problems to solve based on the material. On more practical days, we simply plug a laptop into the projector and show real code; this style works particularly well for concurrency lectures as well as for any discussion sections where you show students code that is relevant for their projects. We don't generally use slides to present material, but have now made a set available for those who prefer that style of presentation.

If you'd like a copy of any of these materials, please drop us an email. We have already shared them with many others around the world.

One last request: if you use the free online chapters, please just **link** to them, instead of making a local copy. This helps us track usage (over 1 million chapters downloaded in the past few years!) and also ensures students get the latest and greatest version.

To Students

If you are a student reading this book, thank you! It is an honor for us to provide some material to help you in your pursuit of knowledge about operating systems. We both think back fondly towards some textbooks of our undergraduate days (e.g., Hennessy and Patterson [HP90], the classic book on computer architecture) and hope this book will become one of those positive memories for you.

You may have noticed this book is free and available online[2]. There is one major reason for this: textbooks are generally too expensive. This book, we hope, is the first of a new wave of free materials to help those in pursuit of their education, regardless of which part of the world they come from or how much they are willing to spend for a book. Failing that, it is one free book, which is better than none.

We also hope, where possible, to point you to the original sources of much of the material in the book: the great papers and persons who have shaped the field of operating systems over the years. Ideas are not pulled out of the air; they come from smart and hard-working people (including numerous Turing-award winners[3]), and thus we should strive to celebrate those ideas and people where possible. In doing so, we hopefully can better understand the revolutions that have taken place, instead of writing texts as if those thoughts have always been present [K62]. Further, perhaps such references will encourage you to dig deeper on your own; reading the famous papers of our field is certainly one of the best ways to learn.

[2] A digression here: "free" in the way we use it here does not mean open source, and it does not mean the book is not copyrighted with the usual protections – it is! What it means is that you can download the chapters and use them to learn about operating systems. Why not an open-source book, just like Linux is an open-source kernel? Well, we believe it is important for a book to have a single voice throughout, and have worked hard to provide such a voice. When you're reading it, the book should kind of feel like a dialogue with the person explaining something to you. Hence, our approach.

[3] The Turing Award is the highest award in Computer Science; it is like the Nobel Prize, except that you have never heard of it.

Acknowledgments

This section will contain thanks to those who helped us put the book together. The important thing for now: **your name could go here!** But, you have to help. So send us some feedback and help debug this book. And you could be famous! Or, at least, have your name in some book.

The people who have helped so far include: Abhirami Senthilkumaran*, Adam Drescher* (WUSTL), Adam Eggum, Aditya Venkataraman, Adriana Iamnitchi and class (USF), Ahmed Fikri*, Ajaykrishna Raghavan, Akiel Khan, Alex Wyler, Ali Razeen (Duke), AmirBehzad Eslami, Anand Mundada, Andrew Valencik (Saint Mary's), Angela Demke Brown (Toronto), B. Brahmananda Reddy (Minnesota), Bala Subrahmanyam Kambala, Benita Bose, Biswajit Mazumder (Clemson), Bobby Jack, Björn Lindberg, Brennan Payne, Brian Gorman, Brian Kroth, Caleb Sumner (Southern Adventist), Cara Lauritzen, Charlotte Kissinger, Chien-Chung Shen (Delaware)*, Christoph Jaeger, Cody Hanson, Dan Soendergaard (U. Aarhus), David Hanle (Grinnell), David Hartman, Deepika Muthukumar, Dheeraj Shetty (North Carolina State), Dorian Arnold (New Mexico), Dustin Metzler, Dustin Passofaro, Eduardo Stelmaszczyk, Emad Sadeghi, Emily Jacobson, Emmett Witchel (Texas), Erik Turk, Ernst Biersack (France), Finn Kuusisto*, Glen Granzow (College of Idaho), Guilherme Baptista, Hamid Reza Ghasemi, Hao Chen, Henry Abbey, Hrishikesh Amur, Huanchen Zhang*, Huseyin Sular, Hugo Diaz, Itai Hass (Toronto), Jake Gillberg, Jakob Olandt, James Perry (U. Michigan-Dearborn)*, Jan Reineke (Universität des Saarlandes), Jay Lim, Jerod Weinman (Grinnell), Jiao Dong (Rutgers), Jingxin Li, Joe Jean (NYU), Joel Kuntz (Saint Mary's), Joel Sommers (Colgate), John Brady (Grinnell), Jonathan Perry (MIT), Jun He, Karl Wallinger, Kartik Singhal, Kaushik Kannan, Kevin Liu*, Lei Tian (U. Nebraska-Lincoln), Leslie Schultz, Liang Yin, Lihao Wang, Martha Ferris, Masashi Kishikawa (Sony), Matt Reichoff, Matty Williams, Meng Huang, Michael Walfish (NYU), Mike Griepentrog, Ming Chen (Stonybrook), Mohammed Alali (Delaware), Murugan Kandaswamy, Natasha Eilbert, Nathan Dipiazza, Nathan Sullivan, Neeraj Badlani (N.C. State), Nelson Gomez, Nghia Huynh (Texas), Nick Weinandt, Patricio Jara, Perry Kivolowitz, Radford Smith, Riccardo Mutschlechner, Ripudaman Singh, Robert Ordòñez and class (Southern Adventist), Rohan Das (Toronto)*, Rohan Pasalkar (Minnesota), Ross Aiken, Ruslan Kiselev, Ryland Herrick, Samer Al-Kiswany, Sandeep Ummadi (Minnesota), Satish Chebrolu (NetApp), Satyanarayana Shanmugam*, Seth Pollen, Sharad Punuganti, Shreevatsa R., Sivaraman Sivaraman*, Srinivasan Thirunarayanan*, Suriyhaprakhas Balaram Sankari, Sy Jin Cheah, Teri Zhao (EMC), Thomas Griebel, Tongxin Zheng, Tony Adkins, Torin Rudeen (Princeton), Tuo Wang, Varun Vats, William Royle (Grinnell), Xiang Peng, Xu Di, Yudong Sun, Yue Zhuo (Texas A&M), Yufui Ren, Zef RosnBrick, Zuyu Zhang. Special thanks to those marked with an asterisk above, who have gone above and beyond in their suggestions for improvement.

In addition, a hearty thanks to Professor Joe Meehean (Lynchburg) for his detailed notes on each chapter, to Professor Jerod Weinman (Grinnell) and his entire class for their incredible booklets, to Professor Chien-Chung Shen (Delaware) for his invaluable and detailed reading and comments, to Adam Drescher (WUSTL) for his careful reading and suggestions, to Glen Granzow (College of Idaho) for his detailed comments and tips, and Michael Walfish (NYU) for his enthusiasm and detailed suggestions for improvement. All have helped these authors immeasur-

ably in the refinement of the materials herein.

Also, many thanks to the hundreds of students who have taken 537 over the years. In particular, the Fall '08 class who encouraged the first written form of these notes (they were sick of not having any kind of textbook to read — pushy students!), and then praised them enough for us to keep going (including one hilarious "ZOMG! You should totally write a textbook!" comment in our course evaluations that year).

A great debt of thanks is also owed to the brave few who took the xv6 project lab course, much of which is now incorporated into the main 537 course. From Spring '09: Justin Cherniak, Patrick Deline, Matt Czech, Tony Gregerson, Michael Griepentrog, Tyler Harter, Ryan Kroiss, Eric Radzikowski, Wesley Reardan, Rajiv Vaidyanathan, and Christopher Waclawik. From Fall '09: Nick Bearson, Aaron Brown, Alex Bird, David Capel, Keith Gould, Tom Grim, Jeffrey Hugo, Brandon Johnson, John Kjell, Boyan Li, James Loethen, Will McCardell, Ryan Szaroletta, Simon Tso, and Ben Yule. From Spring '10: Patrick Blesi, Aidan Dennis-Oehling, Paras Doshi, Jake Friedman, Benjamin Frisch, Evan Hanson, Pikkili Hemanth, Michael Jeung, Alex Langenfeld, Scott Rick, Mike Treffert, Garret Staus, Brennan Wall, Hans Werner, Soo-Young Yang, and Carlos Griffin (almost).

Although they do not directly help with the book, our graduate students have taught us much of what we know about systems. We talk with them regularly while they are at Wisconsin, but they do all the real work — and by telling us about what they are doing, we learn new things every week. This list includes the following collection of current and former students with whom we have published papers; an asterisk marks those who received a Ph.D. under our guidance: Abhishek Rajimwale, Andrew Krioukov, Ao Ma, Brian Forney, Chris Dragga, Deepak Ramamurthi, Florentina Popovici*, Haryadi S. Gunawi*, James Nugent, John Bent*, Jun He, Lanyue Lu, Lakshmi Bairavasundaram*, Laxman Visampalli, Leo Arulraj, Meenali Rungta, Muthian Sivathanu*, Nathan Burnett*, Nitin Agrawal*, Ram Alagappan, Sriram Subramanian*, Stephen Todd Jones*, Suli Yang, Swaminathan Sundararaman*, Swetha Krishnan, Thanh Do*, Thanumalayan S. Pillai, Timothy Denehy*, Tyler Harter, Venkat Venkataramani, Vijay Chidambaram, Vijayan Prabhakaran*, Yiying Zhang*, Yupu Zhang*, Zev Weiss.

A final debt of gratitude is also owed to Aaron Brown, who first took this course many years ago (Spring '09), then took the xv6 lab course (Fall '09), and finally was a graduate teaching assistant for the course for two years or so (Fall '10 through Spring '12). His tireless work has vastly improved the state of the projects (particularly those in xv6 land) and thus has helped better the learning experience for countless undergraduates and graduates here at Wisconsin. As Aaron would say (in his usual succinct manner): "Thx."

Final Words

Yeats famously said "Education is not the filling of a pail but the lighting of a fire." He was right but wrong at the same time[4]. You do have to "fill the pail" a bit, and these notes are certainly here to help with that part of your education; after all, when you go to interview at Google, and they ask you a trick question about how to use semaphores, it might be good to actually know what a semaphore is, right?

But Yeats's larger point is obviously on the mark: the real point of education is to get you interested in something, to learn something more about the subject matter on your own and not just what you have to digest to get a good grade in some class. As one of our fathers (Remzi's dad, Vedat Arpaci) used to say, "Learn beyond the classroom".

We created these notes to spark your interest in operating systems, to read more about the topic on your own, to talk to your professor about all the exciting research that is going on in the field, and even to get involved with that research. It is a great field(!), full of exciting and wonderful ideas that have shaped computing history in profound and important ways. And while we understand this fire won't light for all of you, we hope it does for many, or even a few. Because once that fire is lit, well, that is when you truly become capable of doing something great. And thus the real point of the educational process: to go forth, to study many new and fascinating topics, to learn, to mature, and most importantly, to find something that lights a fire for you.

Andrea and Remzi
Married couple
Professors of Computer Science at the University of Wisconsin
Chief Lighters of Fires, hopefully[5]

[4] If he actually said this; as with many famous quotes, the history of this gem is murky.

[5] If this sounds like we are admitting some past history as arsonists, you are probably missing the point. Probably. If this sounds cheesy, well, that's because it is, but you'll just have to forgive us for that.

References

[CK+08] "The xv6 Operating System"
Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich
From: http://pdos.csail.mit.edu/6.828/2008/index.html
xv6 was developed as a port of the original UNIX *version 6 and represents a beautiful, clean, and simple way to understand a modern operating system.*

[F96] "Six Easy Pieces: Essentials Of Physics Explained By Its Most Brilliant Teacher"
Richard P. Feynman
Basic Books, 1996
This book reprints the six easiest chapters of Feynman's Lectures on Physics, from 1963. If you like Physics, it is a fantastic read.

[HP90] "Computer Architecture a Quantitative Approach" (1st ed.)
David A. Patterson and John L. Hennessy
Morgan-Kaufman, 1990
A book that encouraged each of us at our undergraduate institutions to pursue graduate studies; we later both had the pleasure of working with Patterson, who greatly shaped the foundations of our research careers.

[KR88] "The C Programming Language"
Brian Kernighan and Dennis Ritchie
Prentice-Hall, April 1988
The C programming reference that everyone should have, by the people who invented the language.

[K62] "The Structure of Scientific Revolutions"
Thomas S. Kuhn
University of Chicago Press, 1962
A great and famous read about the fundamentals of the scientific process. Mop-up work, anomaly, crisis, and revolution. We are mostly destined to do mop-up work, alas.

1

A Dialogue on the Book

Professor: *Welcome to this book! It's called* ***Operating Systems in Three Easy Pieces****, and I am here to teach you the things you need to know about operating systems. I am called "Professor"; who are you?*

Student: *Hi Professor! I am called "Student", as you might have guessed. And I am here and ready to learn!*

Professor: *Sounds good. Any questions?*

Student: *Sure! Why is it called "Three Easy Pieces"?*

Professor: *That's an easy one. Well, you see, there are these great lectures on Physics by Richard Feynman...*

Student: *Oh! The guy who wrote "Surely You're Joking, Mr. Feynman", right? Great book! Is this going to be hilarious like that book was?*

Professor: *Um... well, no. That book was great, and I'm glad you've read it. Hopefully this book is more like his notes on Physics. Some of the basics were summed up in a book called "Six Easy Pieces". He was talking about Physics; we're going to do Three Easy Pieces on the fine topic of Operating Systems. This is appropriate, as Operating Systems are about half as hard as Physics.*

Student: *Well, I liked physics, so that is probably good. What are those pieces?*

Professor: *They are the three key ideas we're going to learn about:* ***virtualization****,* ***concurrency****, and* ***persistence****. In learning about these ideas, we'll learn all about how an operating system works, including how it decides what program to run next on a CPU, how it handles memory overload in a virtual memory system, how virtual machine monitors work, how to manage information on disks, and even a little about how to build a distributed system that works when parts have failed. That sort of stuff.*

Student: *I have no idea what you're talking about, really.*

Professor: *Good! That means you are in the right class.*

Student: *I have another question: what's the best way to learn this stuff?*

Professor: *Excellent query! Well, each person needs to figure this out on their*

own, of course, but here is what I would do: go to class, to hear the professor introduce the material. Then, at the end of every week, read these notes, to help the ideas sink into your head a bit better. Of course, some time later (hint: before the exam!), read the notes again to firm up your knowledge. Of course, your professor will no doubt assign some homeworks and projects, so you should do those; in particular, doing projects where you write real code to solve real problems is the best way to put the ideas within these notes into action. As Confucius said...

Student: *Oh, I know! 'I hear and I forget. I see and I remember. I do and I understand.' Or something like that.*

Professor: *(surprised) How did you know what I was going to say?!*

Student: *It seemed to follow. Also, I am a big fan of Confucius, and an even bigger fan of Xunzi, who actually is a better source for this quote*[1].

Professor: *(stunned) Well, I think we are going to get along just fine! Just fine indeed.*

Student: *Professor – just one more question, if I may. What are these dialogues for? I mean, isn't this just supposed to be a book? Why not present the material directly?*

Professor: *Ah, good question, good question! Well, I think it is sometimes useful to pull yourself outside of a narrative and think a bit; these dialogues are those times. So you and I are going to work together to make sense of all of these pretty complex ideas. Are you up for it?*

Student: *So we have to think? Well, I'm up for that. I mean, what else do I have to do anyhow? It's not like I have much of a life outside of this book.*

Professor: *Me neither, sadly. So let's get to work!*

[1] According to this website (http://www.barrypopik.com/index.php/new_york_city/entry/tell_me_and_i_forget_teach_me_and_i_may_remember_involve_me_and_i_will_lear/), Confucian philosopher Xunzi said "Not having heard something is not as good as having heard it; having heard it is not as good as having seen it; having seen it is not as good as knowing it; knowing it is not as good as putting it into practice." Later on, the wisdom got attached to Confucius for some reason. Thanks to Jiao Dong (Rutgers) for telling us!

2

Introduction to Operating Systems

If you are taking an undergraduate operating systems course, you should already have some idea of what a computer program does when it runs. If not, this book (and the corresponding course) is going to be difficult — so you should probably stop reading this book, or run to the nearest bookstore and quickly consume the necessary background material before continuing (both Patt/Patel [PP03] and particularly Bryant/O'Hallaron [BOH10] are pretty great books).

So what happens when a program runs?

Well, a running program does one very simple thing: it executes instructions. Many millions (and these days, even billions) of times every second, the processor **fetches** an instruction from memory, **decodes** it (i.e., figures out which instruction this is), and **executes** it (i.e., it does the thing that it is supposed to do, like add two numbers together, access memory, check a condition, jump to a function, and so forth). After it is done with this instruction, the processor moves on to the next instruction, and so on, and so on, until the program finally completes[1].

Thus, we have just described the basics of the **Von Neumann** model of computing[2]. Sounds simple, right? But in this class, we will be learning that while a program runs, a lot of other wild things are going on with the primary goal of making the system **easy to use**.

There is a body of software, in fact, that is responsible for making it easy to run programs (even allowing you to seemingly run many at the same time), allowing programs to share memory, enabling programs to interact with devices, and other fun stuff like that. That body of software

[1] Of course, modern processors do many bizarre and frightening things underneath the hood to make programs run faster, e.g., executing multiple instructions at once, and even issuing and completing them out of order! But that is not our concern here; we are just concerned with the simple model most programs assume: that instructions seemingly execute one at a time, in an orderly and sequential fashion.

[2] Von Neumann was one of the early pioneers of computing systems. He also did pioneering work on game theory and atomic bombs, and played in the NBA for six years. OK, one of those things isn't true.

THE CRUX OF THE PROBLEM:
HOW TO VIRTUALIZE RESOURCES

One central question we will answer in this book is quite simple: how does the operating system virtualize resources? This is the crux of our problem. *Why* the OS does this is not the main question, as the answer should be obvious: it makes the system easier to use. Thus, we focus on the *how*: what mechanisms and policies are implemented by the OS to attain virtualization? How does the OS do so efficiently? What hardware support is needed?

We will use the "crux of the problem", in shaded boxes such as this one, as a way to call out specific problems we are trying to solve in building an operating system. Thus, within a note on a particular topic, you may find one or more *cruces* (yes, this is the proper plural) which highlight the problem. The details within the chapter, of course, present the solution, or at least the basic parameters of a solution.

is called the **operating system** (**OS**)[3], as it is in charge of making sure the system operates correctly and efficiently in an easy-to-use manner.

The primary way the OS does this is through a general technique that we call **virtualization**. That is, the OS takes a **physical** resource (such as the processor, or memory, or a disk) and transforms it into a more general, powerful, and easy-to-use **virtual** form of itself. Thus, we sometimes refer to the operating system as a **virtual machine**.

Of course, in order to allow users to tell the OS what to do and thus make use of the features of the virtual machine (such as running a program, or allocating memory, or accessing a file), the OS also provides some interfaces (APIs) that you can call. A typical OS, in fact, exports a few hundred **system calls** that are available to applications. Because the OS provides these calls to run programs, access memory and devices, and other related actions, we also sometimes say that the OS provides a **standard library** to applications.

Finally, because virtualization allows many programs to run (thus sharing the CPU), and many programs to concurrently access their own instructions and data (thus sharing memory), and many programs to access devices (thus sharing disks and so forth), the OS is sometimes known as a **resource manager**. Each of the CPU, memory, and disk is a **resource** of the system; it is thus the operating system's role to **manage** those resources, doing so efficiently or fairly or indeed with many other possible goals in mind. To understand the role of the OS a little bit better, let's take a look at some examples.

[3] Another early name for the OS was the **supervisor** or even the **master control program**. Apparently, the latter sounded a little overzealous (see the movie Tron for details) and thus, thankfully, "operating system" caught on instead.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/time.h>
4  #include <assert.h>
5  #include "common.h"
6
7  int
8  main(int argc, char *argv[])
9  {
10     if (argc != 2) {
11         fprintf(stderr, "usage: cpu <string>\n");
12         exit(1);
13     }
14     char *str = argv[1];
15     while (1) {
16         Spin(1);
17         printf("%s\n", str);
18     }
19     return 0;
20 }
```

Figure 2.1: **Simple Example: Code That Loops and Prints (`cpu.c`)**

2.1 Virtualizing the CPU

Figure 2.1 depicts our first program. It doesn't do much. In fact, all it does is call `Spin()`, a function that repeatedly checks the time and returns once it has run for a second. Then, it prints out the string that the user passed in on the command line, and repeats, forever.

Let's say we save this file as `cpu.c` and decide to compile and run it on a system with a single processor (or **CPU** as we will sometimes call it). Here is what we will see:

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
^C
prompt>
```

Not too interesting of a run — the system begins running the program, which repeatedly checks the time until a second has elapsed. Once a second has passed, the code prints the input string passed in by the user (in this example, the letter "A"), and continues. Note the program will run forever; only by pressing "Control-c" (which on UNIX-based systems will terminate the program running in the foreground) can we halt the program.

Now, let's do the same thing, but this time, let's run many different instances of this same program. Figure 2.2 shows the results of this slightly more complicated example.

```
prompt> ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
C
B
D
...
```

Figure 2.2: **Running Many Programs At Once**

Well, now things are getting a little more interesting. Even though we have only one processor, somehow all four of these programs seem to be running at the same time! How does this magic happen?[4]

It turns out that the operating system, with some help from the hardware, is in charge of this **illusion**, i.e., the illusion that the system has a very large number of virtual CPUs. Turning a single CPU (or small set of them) into a seemingly infinite number of CPUs and thus allowing many programs to seemingly run at once is what we call **virtualizing the CPU**, the focus of the first major part of this book.

Of course, to run programs, and stop them, and otherwise tell the OS which programs to run, there need to be some interfaces (APIs) that you can use to communicate your desires to the OS. We'll talk about these APIs throughout this book; indeed, they are the major way in which most users interact with operating systems.

You might also notice that the ability to run multiple programs at once raises all sorts of new questions. For example, if two programs want to run at a particular time, which *should* run? This question is answered by a **policy** of the OS; policies are used in many different places within an OS to answer these types of questions, and thus we will study them as we learn about the basic **mechanisms** that operating systems implement (such as the ability to run multiple programs at once). Hence the role of the OS as a **resource manager**.

[4] Note how we ran four processes at the same time, by using the `&` symbol. Doing so runs a job in the background in the `tcsh` shell, which means that the user is able to immediately issue their next command, which in this case is another program to run. The semi-colon between commands allows us to run multiple programs at the same time in `tcsh`. If you're using a different shell (e.g., `bash`), it works slightly differently; read documentation online for details.

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5
6  int
7  main(int argc, char *argv[])
8  {
9      int *p = malloc(sizeof(int));                    // a1
10     assert(p != NULL);
11     printf("(%d) address pointed to by p: %p\n",
12            getpid(), p);                             // a2
13     *p = 0;                                          // a3
14     while (1) {
15         Spin(1);
16         *p = *p + 1;
17         printf("(%d) p: %d\n", getpid(), *p);        // a4
18     }
19     return 0;
20 }
```

Figure 2.3: **A Program that Accesses Memory (`mem.c`)**

2.2 Virtualizing Memory

Now let's consider memory. The model of **physical memory** presented by modern machines is very simple. Memory is just an array of bytes; to **read** memory, one must specify an **address** to be able to access the data stored there; to **write** (or **update**) memory, one must also specify the data to be written to the given address.

Memory is accessed all the time when a program is running. A program keeps all of its data structures in memory, and accesses them through various instructions, like loads and stores or other explicit instructions that access memory in doing their work. Don't forget that each instruction of the program is in memory too; thus memory is accessed on each instruction fetch.

Let's take a look at a program (in Figure 2.3) that allocates some memory by calling `malloc()`. The output of this program can be found here:

```
prompt> ./mem
(2134) memory address of p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

The program does a couple of things. First, it allocates some memory (line a1). Then, it prints out the address of the memory (a2), and then puts the number zero into the first slot of the newly allocated memory (a3). Finally, it loops, delaying for a second and incrementing the value stored at the address held in `p`. With every print statement, it also prints out what is called the process identifier (the PID) of the running program. This PID is unique per running process.

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) memory address of p: 0x200000
(24114) memory address of p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

Figure 2.4: **Running The Memory Program Multiple Times**

Again, this first result is not too interesting. The newly allocated memory is at address `0x200000`. As the program runs, it slowly updates the value and prints out the result.

Now, we again run multiple instances of this same program to see what happens (Figure 2.4). We see from the example that each running program has allocated memory at the same address (`0x200000`), and yet each seems to be updating the value at `0x200000` independently! It is as if each running program has its own private memory, instead of sharing the same physical memory with other running programs[5].

Indeed, that is exactly what is happening here as the OS is **virtualizing memory**. Each process accesses its own private **virtual address space** (sometimes just called its **address space**), which the OS somehow maps onto the physical memory of the machine. A memory reference within one running program does not affect the address space of other processes (or the OS itself); as far as the running program is concerned, it has physical memory all to itself. The reality, however, is that physical memory is a shared resource, managed by the operating system. Exactly how all of this is accomplished is also the subject of the first part of this book, on the topic of **virtualization**.

2.3 Concurrency

Another main theme of this book is **concurrency**. We use this conceptual term to refer to a host of problems that arise, and must be addressed, when working on many things at once (i.e., concurrently) in the same program. The problems of concurrency arose first within the operating system itself; as you can see in the examples above on virtualization, the OS is juggling many things at once, first running one process, then another, and so forth. As it turns out, doing so leads to some deep and interesting problems.

[5] For this example to work, you need to make sure address-space randomization is disabled; randomization, as it turns out, can be a good defense against certain kinds of security flaws. Read more about it on your own, especially if you want to learn how to break into computer systems via stack-smashing attacks. Not that we would recommend such a thing...

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4
5  volatile int counter = 0;
6  int loops;
7
8  void *worker(void *arg) {
9      int i;
10     for (i = 0; i < loops; i++) {
11         counter++;
12     }
13     return NULL;
14 }
15
16 int
17 main(int argc, char *argv[])
18 {
19     if (argc != 2) {
20         fprintf(stderr, "usage: threads <value>\n");
21         exit(1);
22     }
23     loops = atoi(argv[1]);
24     pthread_t p1, p2;
25     printf("Initial value : %d\n", counter);
26
27     Pthread_create(&p1, NULL, worker, NULL);
28     Pthread_create(&p2, NULL, worker, NULL);
29     Pthread_join(p1, NULL);
30     Pthread_join(p2, NULL);
31     printf("Final value   : %d\n", counter);
32     return 0;
33 }
```

Figure 2.5: **A Multi-threaded Program (`threads.c`)**

Unfortunately, the problems of concurrency are no longer limited just to the OS itself. Indeed, modern **multi-threaded** programs exhibit the same problems. Let us demonstrate with an example of a **multi-threaded** program (Figure 2.5).

Although you might not understand this example fully at the moment (and we'll learn a lot more about it in later chapters, in the section of the book on concurrency), the basic idea is simple. The main program creates two **threads** using `Pthread_create()`[6]. You can think of a thread as a function running within the same memory space as other functions, with more than one of them active at a time. In this example, each thread starts running in a routine called `worker()`, in which it simply increments a counter in a loop for `loops` number of times.

Below is a transcript of what happens when we run this program with the input value for the variable `loops` set to 1000. The value of `loops`

[6]The actual call should be to lower-case `pthread_create()`; the upper-case version is our own wrapper that calls `pthread_create()` and makes sure that the return code indicates that the call succeeded. See the code for details.

THE CRUX OF THE PROBLEM:
HOW TO BUILD CORRECT CONCURRENT PROGRAMS
When there are many concurrently executing threads within the same memory space, how can we build a correctly working program? What primitives are needed from the OS? What mechanisms should be provided by the hardware? How can we use them to solve the problems of concurrency?

determines how many times each of the two workers will increment the shared counter in a loop. When the program is run with the value of `loops` set to 1000, what do you expect the final value of `counter` to be?

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value   : 2000
```

As you probably guessed, when the two threads are finished, the final value of the counter is 2000, as each thread incremented the counter 1000 times. Indeed, when the input value of `loops` is set to N, we would expect the final output of the program to be $2N$. But life is not so simple, as it turns out. Let's run the same program, but with higher values for `loops`, and see what happens:

```
prompt> ./thread 100000
Initial value : 0
Final value   : 143012    // huh??
prompt> ./thread 100000
Initial value : 0
Final value   : 137298    // what the??
```

In this run, when we gave an input value of 100,000, instead of getting a final value of 200,000, we instead first get 143,012. Then, when we run the program a second time, we not only again get the *wrong* value, but also a *different* value than the last time. In fact, if you run the program over and over with high values of `loops`, you may find that sometimes you even get the right answer! So why is this happening?

As it turns out, the reason for these odd and unusual outcomes relate to how instructions are executed, which is one at a time. Unfortunately, a key part of the program above, where the shared counter is incremented, takes three instructions: one to load the value of the counter from memory into a register, one to increment it, and one to store it back into memory. Because these three instructions do not execute **atomically** (all at once), strange things can happen. It is this problem of **concurrency** that we will address in great detail in the second part of this book.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <assert.h>
4  #include <fcntl.h>
5  #include <sys/types.h>
6
7  int
8  main(int argc, char *argv[])
9  {
10     int fd = open("/tmp/file", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU);
11     assert(fd > -1);
12     int rc = write(fd, "hello world\n", 13);
13     assert(rc == 13);
14     close(fd);
15     return 0;
16 }
```

Figure 2.6: **A Program That Does I/O (`io.c`)**

2.4 Persistence

The third major theme of the course is **persistence**. In system memory, data can be easily lost, as devices such as DRAM store values in a **volatile** manner; when power goes away or the system crashes, any data in memory is lost. Thus, we need hardware and software to be able to store data **persistently**; such storage is thus critical to any system as users care a great deal about their data.

The hardware comes in the form of some kind of **input/output** or **I/O** device; in modern systems, a **hard drive** is a common repository for long-lived information, although **solid-state drives** (**SSDs**) are making headway in this arena as well.

The software in the operating system that usually manages the disk is called the **file system**; it is thus responsible for storing any **files** the user creates in a reliable and efficient manner on the disks of the system.

Unlike the abstractions provided by the OS for the CPU and memory, the OS does not create a private, virtualized disk for each application. Rather, it is assumed that often times, users will want to **share** information that is in files. For example, when writing a C program, you might first use an editor (e.g., Emacs[7]) to create and edit the C file (`emacs -nw main.c`). Once done, you might use the compiler to turn the source code into an executable (e.g., `gcc -o main main.c`). When you're finished, you might run the new executable (e.g., `./main`). Thus, you can see how files are shared across different processes. First, Emacs creates a file that serves as input to the compiler; the compiler uses that input file to create a new executable file (in many steps — take a compiler course for details); finally, the new executable is then run. And thus a new program is born!

To understand this better, let's look at some code. Figure 2.6 presents code to create a file (`/tmp/file`) that contains the string "hello world".

[7] You should be using Emacs. If you are using vi, there is probably something wrong with you. If you are using something that is not a real code editor, that is even worse.

THE CRUX OF THE PROBLEM:
HOW TO STORE DATA PERSISTENTLY
The file system is the part of the OS in charge of managing persistent data. What techniques are needed to do so correctly? What mechanisms and policies are required to do so with high performance? How is reliability achieved, in the face of failures in hardware and software?

To accomplish this task, the program makes three calls into the operating system. The first, a call to `open()`, opens the file and creates it; the second, `write()`, writes some data to the file; the third, `close()`, simply closes the file thus indicating the program won't be writing any more data to it. These **system calls** are routed to the part of the operating system called the **file system**, which then handles the requests and returns some kind of error code to the user.

You might be wondering what the OS does in order to actually write to disk. We would show you but you'd have to promise to close your eyes first; it is that unpleasant. The file system has to do a fair bit of work: first figuring out where on disk this new data will reside, and then keeping track of it in various structures the file system maintains. Doing so requires issuing I/O requests to the underlying storage device, to either read existing structures or update (write) them. As anyone who has written a **device driver**[8] knows, getting a device to do something on your behalf is an intricate and detailed process. It requires a deep knowledge of the low-level device interface and its exact semantics. Fortunately, the OS provides a standard and simple way to access devices through its system calls. Thus, the OS is sometimes seen as a **standard library**.

Of course, there are many more details in how devices are accessed, and how file systems manage data persistently atop said devices. For performance reasons, most file systems first delay such writes for a while, hoping to batch them into larger groups. To handle the problems of system crashes during writes, most file systems incorporate some kind of intricate write protocol, such as **journaling** or **copy-on-write**, carefully ordering writes to disk to ensure that if a failure occurs during the write sequence, the system can recover to reasonable state afterwards. To make different common operations efficient, file systems employ many different data structures and access methods, from simple lists to complex b-trees. If all of this doesn't make sense yet, good! We'll be talking about all of this quite a bit more in the third part of this book on **persistence**, where we'll discuss devices and I/O in general, and then disks, RAIDs, and file systems in great detail.

[8] A device driver is some code in the operating system that knows how to deal with a specific device. We will talk more about devices and device drivers later.

2.5 Design Goals

So now you have some idea of what an OS actually does: it takes physical **resources**, such as a CPU, memory, or disk, and **virtualizes** them. It handles tough and tricky issues related to **concurrency**. And it stores files **persistently**, thus making them safe over the long-term. Given that we want to build such a system, we want to have some goals in mind to help focus our design and implementation and make trade-offs as necessary; finding the right set of trade-offs is a key to building systems.

One of the most basic goals is to build up some **abstractions** in order to make the system convenient and easy to use. Abstractions are fundamental to everything we do in computer science. Abstraction makes it possible to write a large program by dividing it into small and understandable pieces, to write such a program in a high-level language like C[9] without thinking about assembly, to write code in assembly without thinking about logic gates, and to build a processor out of gates without thinking too much about transistors. Abstraction is so fundamental that sometimes we forget its importance, but we won't here; thus, in each section, we'll discuss some of the major abstractions that have developed over time, giving you a way to think about pieces of the OS.

One goal in designing and implementing an operating system is to provide high **performance**; another way to say this is our goal is to **minimize the overheads** of the OS. Virtualization and making the system easy to use are well worth it, but not at any cost; thus, we must strive to provide virtualization and other OS features without excessive overheads. These overheads arise in a number of forms: extra time (more instructions) and extra space (in memory or on disk). We'll seek solutions that minimize one or the other or both, if possible. Perfection, however, is not always attainable, something we will learn to notice and (where appropriate) tolerate.

Another goal will be to provide **protection** between applications, as well as between the OS and applications. Because we wish to allow many programs to run at the same time, we want to make sure that the malicious or accidental bad behavior of one does not harm others; we certainly don't want an application to be able to harm the OS itself (as that would affect *all* programs running on the system). Protection is at the heart of one of the main principles underlying an operating system, which is that of **isolation**; isolating processes from one another is the key to protection and thus underlies much of what an OS must do.

The operating system must also run non-stop; when it fails, *all* applications running on the system fail as well. Because of this dependence, operating systems often strive to provide a high degree of **reliability**. As operating systems grow evermore complex (sometimes containing millions of lines of code), building a reliable operating system is quite a chal-

[9] Some of you might object to calling C a high-level language. Remember this is an OS course, though, where we're simply happy not to have to code in assembly all the time!

lenge — and indeed, much of the on-going research in the field (including some of our own work [BS+09, SS+10]) focuses on this exact problem.

Other goals make sense: **energy-efficiency** is important in our increasingly green world; **security** (an extension of protection, really) against malicious applications is critical, especially in these highly-networked times; **mobility** is increasingly important as OSes are run on smaller and smaller devices. Depending on how the system is used, the OS will have different goals and thus likely be implemented in at least slightly different ways. However, as we will see, many of the principles we will present on how to build an OS are useful on a range of different devices.

2.6 Some History

Before closing this introduction, let us present a brief history of how operating systems developed. Like any system built by humans, good ideas accumulated in operating systems over time, as engineers learned what was important in their design. Here, we discuss a few major developments. For a richer treatment, see Brinch Hansen's excellent history of operating systems [BH00].

Early Operating Systems: Just Libraries

In the beginning, the operating system didn't do too much. Basically, it was just a set of libraries of commonly-used functions; for example, instead of having each programmer of the system write low-level I/O handling code, the "OS" would provide such APIs, and thus make life easier for the developer.

Usually, on these old mainframe systems, one program ran at a time, as controlled by a human operator. Much of what you think a modern OS would do (e.g., deciding what order to run jobs in) was performed by this operator. If you were a smart developer, you would be nice to this operator, so that they might move your job to the front of the queue.

This mode of computing was known as **batch** processing, as a number of jobs were set up and then run in a "batch" by the operator. Computers, as of that point, were not used in an interactive manner, because of cost: it was simply too expensive to let a user sit in front of the computer and use it, as most of the time it would just sit idle then, costing the facility hundreds of thousands of dollars per hour [BH00].

Beyond Libraries: Protection

In moving beyond being a simple library of commonly-used services, operating systems took on a more central role in managing machines. One important aspect of this was the realization that code run on behalf of the OS was special; it had control of devices and thus should be treated differently than normal application code. Why is this? Well, imagine if you

allowed any application to read from anywhere on the disk; the notion of privacy goes out the window, as any program could read any file. Thus, implementing a **file system** (to manage your files) as a library makes little sense. Instead, something else was needed.

Thus, the idea of a **system call** was invented, pioneered by the Atlas computing system [K+61,L78]. Instead of providing OS routines as a library (where you just make a **procedure call** to access them), the idea here was to add a special pair of hardware instructions and hardware state to make the transition into the OS a more formal, controlled process.

The key difference between a system call and a procedure call is that a system call transfers control (i.e., jumps) into the OS while simultaneously raising the **hardware privilege level**. User applications run in what is referred to as **user mode** which means the hardware restricts what applications can do; for example, an application running in user mode can't typically initiate an I/O request to the disk, access any physical memory page, or send a packet on the network. When a system call is initiated (usually through a special hardware instruction called a **trap**), the hardware transfers control to a pre-specified **trap handler** (that the OS set up previously) and simultaneously raises the privilege level to **kernel mode**. In kernel mode, the OS has full access to the hardware of the system and thus can do things like initiate an I/O request or make more memory available to a program. When the OS is done servicing the request, it passes control back to the user via a special **return-from-trap** instruction, which reverts to user mode while simultaneously passing control back to where the application left off.

The Era of Multiprogramming

Where operating systems really took off was in the era of computing beyond the mainframe, that of the **minicomputer**. Classic machines like the PDP family from Digital Equipment made computers hugely more affordable; thus, instead of having one mainframe per large organization, now a smaller collection of people within an organization could likely have their own computer. Not surprisingly, one of the major impacts of this drop in cost was an increase in developer activity; more smart people got their hands on computers and thus made computer systems do more interesting and beautiful things.

In particular, **multiprogramming** became commonplace due to the desire to make better use of machine resources. Instead of just running one job at a time, the OS would load a number of jobs into memory and switch rapidly between them, thus improving CPU utilization. This switching was particularly important because I/O devices were slow; having a program wait on the CPU while its I/O was being serviced was a waste of CPU time. Instead, why not switch to another job and run it for a while?

The desire to support multiprogramming and overlap in the presence of I/O and interrupts forced innovation in the conceptual development of operating systems along a number of directions. Issues such as **memory**

protection became important; we wouldn't want one program to be able to access the memory of another program. Understanding how to deal with the **concurrency** issues introduced by multiprogramming was also critical; making sure the OS was behaving correctly despite the presence of interrupts is a great challenge. We will study these issues and related topics later in the book.

One of the major practical advances of the time was the introduction of the UNIX operating system, primarily thanks to Ken Thompson (and Dennis Ritchie) at Bell Labs (yes, the phone company). UNIX took many good ideas from different operating systems (particularly from Multics [O72], and some from systems like TENEX [B+72] and the Berkeley Time-Sharing System [S+68]), but made them simpler and easier to use. Soon this team was shipping tapes containing UNIX source code to people around the world, many of whom then got involved and added to the system themselves; see the **Aside** (next page) for more detail[10].

The Modern Era

Beyond the minicomputer came a new type of machine, cheaper, faster, and for the masses: the **personal computer**, or **PC** as we call it today. Led by Apple's early machines (e.g., the Apple II) and the IBM PC, this new breed of machine would soon become the dominant force in computing, as their low-cost enabled one machine per desktop instead of a shared minicomputer per workgroup.

Unfortunately, for operating systems, the PC at first represented a great leap backwards, as early systems forgot (or never knew of) the lessons learned in the era of minicomputers. For example, early operating systems such as **DOS** (the **Disk Operating System**, from **Microsoft**) didn't think memory protection was important; thus, a malicious (or perhaps just a poorly-programmed) application could scribble all over memory. The first generations of the **Mac OS** (v9 and earlier) took a cooperative approach to job scheduling; thus, a thread that accidentally got stuck in an infinite loop could take over the entire system, forcing a reboot. The painful list of OS features missing in this generation of systems is long, too long for a full discussion here.

Fortunately, after some years of suffering, the old features of minicomputer operating systems started to find their way onto the desktop. For example, Mac OS X has UNIX at its core, including all of the features one would expect from such a mature system. Windows has similarly adopted many of the great ideas in computing history, starting in particular with Windows NT, a great leap forward in Microsoft OS technology. Even today's cell phones run operating systems (such as Linux) that are much more like what a minicomputer ran in the 1970s than what a PC

[10] We'll use asides and other related text boxes to call attention to various items that don't quite fit the main flow of the text. Sometimes, we'll even use them just to make a joke, because why not have a little fun along the way? Yes, many of the jokes are bad.

ASIDE: **THE IMPORTANCE OF UNIX**

It is difficult to overstate the importance of UNIX in the history of operating systems. Influenced by earlier systems (in particular, the famous **Multics** system from MIT), UNIX brought together many great ideas and made a system that was both simple and powerful.

Underlying the original "Bell Labs" UNIX was the unifying principle of building small powerful programs that could be connected together to form larger workflows. The **shell**, where you type commands, provided primitives such as **pipes** to enable such meta-level programming, and thus it became easy to string together programs to accomplish a bigger task. For example, to find lines of a text file that have the word "foo" in them, and then to count how many such lines exist, you would type: `grep foo file.txt|wc -l`, thus using the `grep` and `wc` (word count) programs to achieve your task.

The UNIX environment was friendly for programmers and developers alike, also providing a compiler for the new **C programming language**. Making it easy for programmers to write their own programs, as well as share them, made UNIX enormously popular. And it probably helped a lot that the authors gave out copies for free to anyone who asked, an early form of **open-source software**.

Also of critical importance was the accessibility and readability of the code. Having a beautiful, small kernel written in C invited others to play with the kernel, adding new and cool features. For example, an enterprising group at Berkeley, led by **Bill Joy**, made a wonderful distribution (the **Berkeley Systems Distribution**, or **BSD**) which had some advanced virtual memory, file system, and networking subsystems. Joy later co-founded **Sun Microsystems**.

Unfortunately, the spread of UNIX was slowed a bit as companies tried to assert ownership and profit from it, an unfortunate (but common) result of lawyers getting involved. Many companies had their own variants: **SunOS** from Sun Microsystems, **AIX** from IBM, **HPUX** (a.k.a. "H-Pucks") from HP, and **IRIX** from SGI. The legal wrangling among AT&T/Bell Labs and these other players cast a dark cloud over UNIX, and many wondered if it would survive, especially as Windows was introduced and took over much of the PC market...

ran in the 1980s (thank goodness); it is good to see that the good ideas developed in the heyday of OS development have found their way into the modern world. Even better is that these ideas continue to develop, providing more features and making modern systems even better for users and applications.

ASIDE: **AND THEN CAME LINUX**

Fortunately for UNIX, a young Finnish hacker named **Linus Torvalds** decided to write his own version of UNIX which borrowed heavily on the principles and ideas behind the original system, but not from the code base, thus avoiding issues of legality. He enlisted help from many others around the world, and soon **Linux** was born (as well as the modern open-source software movement).

As the internet era came into place, most companies (such as Google, Amazon, Facebook, and others) chose to run Linux, as it was free and could be readily modified to suit their needs; indeed, it is hard to imagine the success of these new companies had such a system not existed. As smart phones became a dominant user-facing platform, Linux found a stronghold there too (via Android), for many of the same reasons. And Steve Jobs took his UNIX-based **NeXTStep** operating environment with him to Apple, thus making UNIX popular on desktops (though many users of Apple technology are probably not even aware of this fact). And thus UNIX lives on, more important today than ever before. The computing gods, if you believe in them, should be thanked for this wonderful outcome.

2.7 Summary

Thus, we have an introduction to the OS. Today's operating systems make systems relatively easy to use, and virtually all operating systems you use today have been influenced by the developments we will discuss throughout the book.

Unfortunately, due to time constraints, there are a number of parts of the OS we won't cover in the book. For example, there is a lot of **networking** code in the operating system; we leave it to you to take the networking class to learn more about that. Similarly, **graphics** devices are particularly important; take the graphics course to expand your knowledge in that direction. Finally, some operating system books talk a great deal about **security**; we will do so in the sense that the OS must provide protection between running programs and give users the ability to protect their files, but we won't delve into deeper security issues that one might find in a security course.

However, there are many important topics that we will cover, including the basics of virtualization of the CPU and memory, concurrency, and persistence via devices and file systems. Don't worry! While there is a lot of ground to cover, most of it is quite cool, and at the end of the road, you'll have a new appreciation for how computer systems really work. Now get to work!

References

[BS+09] "Tolerating File-System Mistakes with EnvyFS"
Lakshmi N. Bairavasundaram, Swaminathan Sundararaman, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
USENIX '09, San Diego, CA, June 2009
A fun paper about using multiple file systems at once to tolerate a mistake in any one of them.

[BH00] "The Evolution of Operating Systems"
P. Brinch Hansen
In Classic Operating Systems: From Batch Processing to Distributed Systems
Springer-Verlag, New York, 2000
This essay provides an intro to a wonderful collection of papers about historically significant systems.

[B+72] "TENEX, A Paged Time Sharing System for the PDP-10"
Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, Raymond S. Tomlinson
CACM, Volume 15, Number 3, March 1972
TENEX has much of the machinery found in modern operating systems; read more about it to see how much innovation was already in place in the early 1970's.

[B75] "The Mythical Man-Month"
Fred Brooks
Addison-Wesley, 1975
A classic text on software engineering; well worth the read.

[BOH10] "Computer Systems: A Programmer's Perspective"
Randal E. Bryant and David R. O'Hallaron
Addison-Wesley, 2010
Another great intro to how computer systems work. Has a little bit of overlap with this book — so if you'd like, you can skip the last few chapters of that book, or simply read them to get a different perspective on some of the same material. After all, one good way to build up your own knowledge is to hear as many other perspectives as possible, and then develop your own opinion and thoughts on the matter. You know, by thinking!

[K+61] "One-Level Storage System"
T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner
IRE Transactions on Electronic Computers, April 1962
The Atlas pioneered much of what you see in modern systems. However, this paper is not the best read. If you were to only read one, you might try the historical perspective below [L78].

[L78] "The Manchester Mark I and Atlas: A Historical Perspective"
S. H. Lavington
Communications of the ACM archive
Volume 21, Issue 1 (January 1978), pages 4-12
A nice piece of history on the early development of computer systems and the pioneering efforts of the Atlas. Of course, one could go back and read the Atlas papers themselves, but this paper provides a great overview and adds some historical perspective.

[O72] "The Multics System: An Examination of its Structure"
Elliott Organick, 1972
A great overview of Multics. So many good ideas, and yet it was an over-designed system, shooting for too much, and thus never really worked as expected. A classic example of what Fred Brooks would call the "second-system effect" [B75].

[PP03] "Introduction to Computing Systems:
From Bits and Gates to C and Beyond"
Yale N. Patt and Sanjay J. Patel
McGraw-Hill, 2003
One of our favorite intro to computing systems books. Starts at transistors and gets you all the way up to C; the early material is particularly great.

[RT74] "The UNIX Time-Sharing System"
Dennis M. Ritchie and Ken Thompson
CACM, Volume 17, Number 7, July 1974, pages 365-375
A great summary of UNIX written as it was taking over the world of computing, by the people who wrote it.

[S68] "SDS 940 Time-Sharing System"
Scientific Data Systems Inc.
TECHNICAL MANUAL, SDS 90 11168 August 1968
Available: http://goo.gl/EN0Zrn
Yes, a technical manual was the best we could find. But it is fascinating to read these old system documents, and see how much was already in place in the late 1960's. One of the minds behind the Berkeley Time-Sharing System (which eventually became the SDS system) was Butler Lampson, who later won a Turing award for his contributions in systems.

[SS+10] "Membrane: Operating System Support for Restartable File Systems"
Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Michael M. Swift
FAST '10, San Jose, CA, February 2010
The great thing about writing your own class notes: you can advertise your own research. But this paper is actually pretty neat — when a file system hits a bug and crashes, Membrane auto-magically restarts it, all without applications or the rest of the system being affected.

Part I

Virtualization

3

A Dialogue on Virtualization

Professor: *And thus we reach the first of our three pieces on operating systems:* ***virtualization****.*

Student: *But what is virtualization, oh noble professor?*

Professor: *Imagine we have a peach.*

Student: *A peach? (incredulous)*

Professor: *Yes, a peach. Let us call that the* ***physical*** *peach. But we have many eaters who would like to eat this peach. What we would like to present to each eater is their own peach, so that they can be happy. We call the peach we give eaters* ***virtual*** *peaches; we somehow create many of these virtual peaches out of the one physical peach. And the important thing: in this illusion, it looks to each eater like they have a physical peach, but in reality they don't.*

Student: *So you are sharing the peach, but you don't even know it?*

Professor: *Right! Exactly.*

Student: *But there's only one peach.*

Professor: *Yes. And...?*

Student: *Well, if I was sharing a peach with somebody else, I think I would notice.*

Professor: *Ah yes! Good point. But that is the thing with many eaters; most of the time they are napping or doing something else, and thus, you can snatch that peach away and give it to someone else for a while. And thus we create the illusion of many virtual peaches, one peach for each person!*

Student: *Sounds like a bad campaign slogan. You are talking about computers, right Professor?*

Professor: *Ah, young grasshopper, you wish to have a more concrete example. Good idea! Let us take the most basic of resources, the CPU. Assume there is one physical CPU in a system (though now there are often two or four or more). What virtualization does is take that single CPU and make it look like many virtual CPUs to the applications running on the system. Thus, while each application*

thinks it has its own CPU to use, there is really only one. And thus the OS has created a beautiful illusion: it has virtualized the CPU.

Student: *Wow! That sounds like magic. Tell me more! How does that work?*

Professor: *In time, young student, in good time. Sounds like you are ready to begin.*

Student: *I am! Well, sort of. I must admit, I'm a little worried you are going to start talking about peaches again.*

Professor: *Don't worry too much; I don't even like peaches. And thus we begin...*

4

The Abstraction: The Process

In this chapter, we discuss one of the most fundamental abstractions that the OS provides to users: the **process**. The definition of a process, informally, is quite simple: it is a **running program** [V+65,BH70]. The program itself is a lifeless thing: it just sits there on the disk, a bunch of instructions (and maybe some static data), waiting to spring into action. It is the operating system that takes these bytes and gets them running, transforming the program into something useful.

It turns out that one often wants to run more than one program at once; for example, consider your desktop or laptop where you might like to run a web browser, mail program, a game, a music player, and so forth. In fact, a typical system may be seemingly running tens or even hundreds of processes at the same time. Doing so makes the system easy to use, as one never need be concerned with whether a CPU is available; one simply runs programs. Hence our challenge:

THE CRUX OF THE PROBLEM:
HOW TO PROVIDE THE ILLUSION OF MANY CPUS?
Although there are only a few physical CPUs available, how can the OS provide the illusion of a nearly-endless supply of said CPUs?

The OS creates this illusion by **virtualizing** the CPU. By running one process, then stopping it and running another, and so forth, the OS can promote the illusion that many virtual CPUs exist when in fact there is only one physical CPU (or a few). This basic technique, known as **time sharing** of the CPU, allows users to run as many concurrent processes as they would like; the potential cost is performance, as each will run more slowly if the CPU(s) must be shared.

To implement virtualization of the CPU, and to implement it well, the OS will need both some low-level machinery as well as some high-level intelligence. We call the low-level machinery **mechanisms**; mechanisms are low-level methods or protocols that implement a needed piece

TIP: USE TIME SHARING (AND SPACE SHARING)
Time sharing is one of the most basic techniques used by an OS to share a resource. By allowing the resource to be used for a little while by one entity, and then a little while by another, and so forth, the resource in question (e.g., the CPU, or a network link) can be shared by many. The natural counterpart of time sharing is **space sharing**, where a resource is divided (in space) among those who wish to use it. For example, disk space is naturally a space-shared resource, as once a block is assigned to a file, it is not likely to be assigned to another file until the user deletes it.

of functionality. For example, we'll learn later how to implement a **context switch**, which gives the OS the ability to stop running one program and start running another on a given CPU; this **time-sharing** mechanism is employed by all modern OSes.

On top of these mechanisms resides some of the intelligence in the OS, in the form of **policies**. Policies are algorithms for making some kind of decision within the OS. For example, given a number of possible programs to run on a CPU, which program should the OS run? A **scheduling policy** in the OS will make this decision, likely using historical information (e.g., which program has run more over the last minute?), workload knowledge (e.g., what types of programs are run), and performance metrics (e.g., is the system optimizing for interactive performance, or throughput?) to make its decision.

4.1 The Abstraction: A Process

The abstraction provided by the OS of a running program is something we will call a **process**. As we said above, a process is simply a running program; at any instant in time, we can summarize a process by taking an inventory of the different pieces of the system it accesses or affects during the course of its execution.

To understand what constitutes a process, we thus have to understand its **machine state**: what a program can read or update when it is running. At any given time, what parts of the machine are important to the execution of this program?

One obvious component of machine state that comprises a process is its *memory*. Instructions lie in memory; the data that the running program reads and writes sits in memory as well. Thus the memory that the process can address (called its **address space**) is part of the process.

Also part of the process's machine state are *registers*; many instructions explicitly read or update registers and thus clearly they are important to the execution of the process.

Note that there are some particularly special registers that form part of this machine state. For example, the **program counter** (**PC**) (sometimes called the **instruction pointer** or **IP**) tells us which instruction of the pro-

TIP: SEPARATE POLICY AND MECHANISM
In many operating systems, a common design paradigm is to separate high-level policies from their low-level mechanisms [L+75]. You can think of the mechanism as providing the answer to a *how* question about a system; for example, *how* does an operating system perform a context switch? The policy provides the answer to a *which* question; for example, *which* process should the operating system run right now? Separating the two allows one easily to change policies without having to rethink the mechanism and is thus a form of **modularity**, a general software design principle.

gram is currently being executed; similarly a **stack pointer** and associated **frame pointer** are used to manage the stack for function parameters, local variables, and return addresses.

Finally, programs often access persistent storage devices too. Such *I/O information* might include a list of the files the process currently has open.

4.2 Process API

Though we defer discussion of a real process API until a subsequent chapter, here we first give some idea of what must be included in any interface of an operating system. These APIs, in some form, are available on any modern operating system.

- **Create:** An operating system must include some method to create new processes. When you type a command into the shell, or double-click on an application icon, the OS is invoked to create a new process to run the program you have indicated.
- **Destroy:** As there is an interface for process creation, systems also provide an interface to destroy processes forcefully. Of course, many processes will run and just exit by themselves when complete; when they don't, however, the user may wish to kill them, and thus an interface to halt a runaway process is quite useful.
- **Wait:** Sometimes it is useful to wait for a process to stop running; thus some kind of waiting interface is often provided.
- **Miscellaneous Control:** Other than killing or waiting for a process, there are sometimes other controls that are possible. For example, most operating systems provide some kind of method to suspend a process (stop it from running for a while) and then resume it (continue it running).
- **Status:** There are usually interfaces to get some status information about a process as well, such as how long it has run for, or what state it is in.

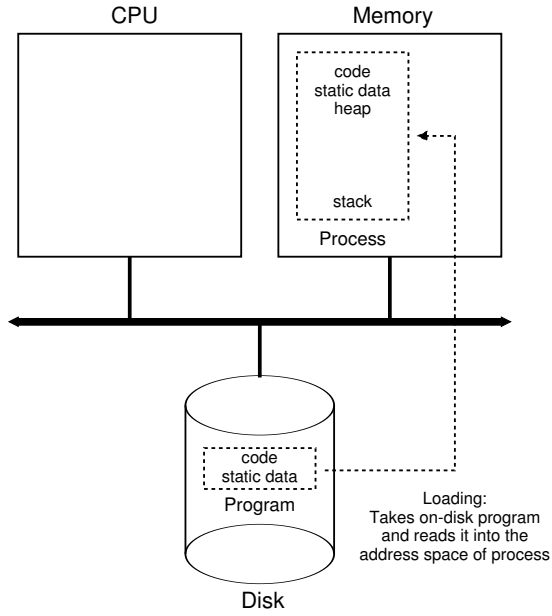


Figure 4.1: **Loading: From Program To Process**

4.3 Process Creation: A Little More Detail

One mystery that we should unmask a bit is how programs are transformed into processes. Specifically, how does the OS get a program up and running? How does process creation actually work?

The first thing that the OS must do to run a program is to **load** its code and any static data (e.g., initialized variables) into memory, into the address space of the process. Programs initially reside on **disk** (or, in some modern systems, **flash-based SSDs**) in some kind of **executable format**; thus, the process of loading a program and static data into memory requires the OS to read those bytes from disk and place them in memory somewhere (as shown in Figure 4.1).

In early (or simple) operating systems, the loading process is done **eagerly**, i.e., all at once before running the program; modern OSes perform the process **lazily**, i.e., by loading pieces of code or data only as they are needed during program execution. To truly understand how lazy loading of pieces of code and data works, you'll have to understand more about the machinery of **paging** and **swapping**, topics we'll cover in the future when we discuss the virtualization of memory. For now, just remember that before running anything, the OS clearly must do some work to get the important program bits from disk into memory.

Once the code and static data are loaded into memory, there are a few other things the OS needs to do before running the process. Some memory must be allocated for the program's **run-time stack** (or just **stack**). As you should likely already know, C programs use the stack for local variables, function parameters, and return addresses; the OS allocates this memory and gives it to the process. The OS will also likely initialize the stack with arguments; specifically, it will fill in the parameters to the `main()` function, i.e., `argc` and the `argv` array.

The OS may also allocate some memory for the program's **heap**. In C programs, the heap is used for explicitly requested dynamically-allocated data; programs request such space by calling `malloc()` and free it explicitly by calling `free()`. The heap is needed for data structures such as linked lists, hash tables, trees, and other interesting data structures. The heap will be small at first; as the program runs, and requests more memory via the `malloc()` library API, the OS may get involved and allocate more memory to the process to help satisfy such calls.

The OS will also do some other initialization tasks, particularly as related to input/output (I/O). For example, in UNIX systems, each process by default has three open **file descriptors**, for standard input, output, and error; these descriptors let programs easily read input from the terminal as well as print output to the screen. We'll learn more about I/O, file descriptors, and the like in the third part of the book on **persistence**.

By loading the code and static data into memory, by creating and initializing a stack, and by doing other work as related to I/O setup, the OS has now (finally) set the stage for program execution. It thus has one last task: to start the program running at the entry point, namely `main()`. By jumping to the `main()` routine (through a specialized mechanism that we will discuss next chapter), the OS transfers control of the CPU to the newly-created process, and thus the program begins its execution.

4.4 Process States

Now that we have some idea of what a process is (though we will continue to refine this notion), and (roughly) how it is created, let us talk about the different **states** a process can be in at a given time. The notion that a process can be in one of these states arose in early computer systems [DV66,V+65]. In a simplified view, a process can be in one of three states:

- **Running**: In the running state, a process is running on a processor. This means it is executing instructions.
- **Ready**: In the ready state, a process is ready to run but for some reason the OS has chosen not to run it at this given moment.
- **Blocked**: In the blocked state, a process has performed some kind of operation that makes it not ready to run until some other event takes place. A common example: when a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

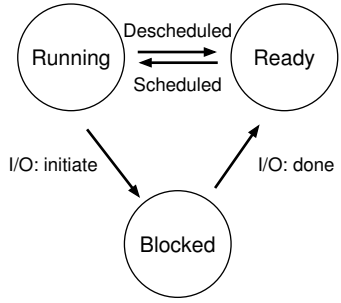


Figure 4.2: **Process: State Transitions**

If we were to map these states to a graph, we would arrive at the diagram in Figure 4.2. As you can see in the diagram, a process can be moved between the ready and running states at the discretion of the OS. Being moved from ready to running means the process has been **scheduled**; being moved from running to ready means the process has been **descheduled**. Once a process has become blocked (e.g., by initiating an I/O operation), the OS will keep it as such until some event occurs (e.g., I/O completion); at that point, the process moves to the ready state again (and potentially immediately to running again, if the OS so decides).

Let's look at an example of how two processes might transition through some of these states. First, imagine two processes running, each of which only use the CPU (they do no I/O). In this case, a trace of the state of each process might look like this (Figure 4.3).

Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process_0 now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	Process_1 now done

Figure 4.3: **Tracing Process State: CPU Only**

In this next example, the first process issues an I/O after running for some time. At that point, the process is blocked, giving the other process a chance to run. Figure 4.4 shows a trace of this scenario.

More specifically, Process_0 initiates an I/O and becomes blocked waiting for it to complete; processes become blocked, for example, when read-

Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process_0 initiates I/O
4	Blocked	Running	Process_0 is blocked,
5	Blocked	Running	so Process_1 runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process_1 now done
9	Running	–	
10	Running	–	Process_0 now done

Figure 4.4: **Tracing Process State: CPU and I/O**

ing from a disk or waiting for a packet from a network. The OS recognizes Process_0 is not using the CPU and starts running Process_1. While Process_1 is running, the I/O completes, moving Process_0 back to ready. Finally, Process_1 finishes, and Process_0 runs and then is done.

Note that there are many decisions the OS must make, even in this simple example. First, the system had to decide to run Process_1 while Process_0 issued an I/O; doing so improves resource utilization by keeping the CPU busy. Second, the system decided not to switch back to Process_0 when its I/O completed; it is not clear if this is a good decision or not. What do you think? These types of decisions are made by the OS **scheduler**, a topic we will discuss a few chapters in the future.

4.5 Data Structures

The OS is a program, and like any program, it has some key data structures that track various relevant pieces of information. To track the state of each process, for example, the OS likely will keep some kind of **process list** for all processes that are ready, as well as some additional information to track which process is currently running. The OS must also track, in some way, blocked processes; when an I/O event completes, the OS should make sure to wake the correct process and ready it to run again.

Figure 4.5 shows what type of information an OS needs to track about each process in the xv6 kernel [CK+08]. Similar process structures exist in "real" operating systems such as Linux, Mac OS X, or Windows; look them up and see how much more complex they are.

From the figure, you can see a couple of important pieces of information the OS tracks about a process. The **register context** will hold, for a stopped process, the contents of its registers. When a process is stopped, its registers will be saved to this memory location; by restoring these registers (i.e., placing their values back into the actual physical registers), the OS can resume running the process. We'll learn more about this technique known as a **context switch** in future chapters.

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If non-zero, sleeping on chan
  int killed;                 // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

Figure 4.5: **The xv6 Proc Structure**

You can also see from the figure that there are some other states a process can be in, beyond running, ready, and blocked. Sometimes a system will have an **initial** state that the process is in when it is being created. Also, a process could be placed in a **final** state where it has exited but has not yet been cleaned up (in UNIX-based systems, this is called the **zombie** state[1]). This final state can be useful as it allows other processes (usually the **parent** that created the process) to examine the return code of the process and see if the just-finished process executed successfully (usually, programs return zero in UNIX-based systems when they have accomplished a task successfully, and non-zero otherwise). When finished, the parent will make one final call (e.g., `wait()`) to wait for the completion of the child, and to also indicate to the OS that it can clean up any relevant data structures that referred to the now-extinct process.

[1] Yes, the zombie state. Just like real zombies, these zombies are relatively easy to kill. However, different techniques are usually recommended.

ASIDE: **DATA STRUCTURE — THE PROCESS LIST**
Operating systems are replete with various important **data structures** that we will discuss in these notes. The **process list** is the first such structure. It is one of the simpler ones, but certainly any OS that has the ability to run multiple programs at once will have something akin to this structure in order to keep track of all the running programs in the system. Sometimes people refer to the individual structure that stores information about a process as a **Process Control Block** (**PCB**), a fancy way of talking about a C structure that contains information about each process.

4.6 Summary

We have introduced the most basic abstraction of the OS: the process. It is quite simply viewed as a running program. With this conceptual view in mind, we will now move on to the nitty-gritty: the low-level mechanisms needed to implement processes, and the higher-level policies required to schedule them in an intelligent way. By combining mechanisms and policies, we will build up our understanding of how an operating system virtualizes the CPU.

References

[BH70] "The Nucleus of a Multiprogramming System"
Per Brinch Hansen
Communications of the ACM, Volume 13, Number 4, April 1970
*This paper introduces one of the first **microkernels** in operating systems history, called Nucleus. The idea of smaller, more minimal systems is a theme that rears its head repeatedly in OS history; it all began with Brinch Hansen's work described herein.*

[CK+08] "The xv6 Operating System"
Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich
From: http://pdos.csail.mit.edu/6.828/2008/index.html
The coolest real and little OS in the world. Download and play with it to learn more about the details of how operating systems actually work.

[DV66] "Programming Semantics for Multiprogrammed Computations"
Jack B. Dennis and Earl C. Van Horn
Communications of the ACM, Volume 9, Number 3, March 1966
This paper defined many of the early terms and concepts around building multiprogrammed systems.

[L+75] "Policy/mechanism separation in Hydra"
R. Levin, E. Cohen, W. Corwin, F. Pollack, W. Wulf
SOSP 1975
An early paper about how to structure operating systems in a research OS known as Hydra. While Hydra never became a mainstream OS, some of its ideas influenced OS designers.

[V+65] "Structure of the Multics Supervisor"
V.A. Vyssotsky, F. J. Corbato, R. M. Graham
Fall Joint Computer Conference, 1965
An early paper on Multics, which described many of the basic ideas and terms that we find in modern systems. Some of the vision behind computing as a utility are finally being realized in modern cloud systems.

Homework

ASIDE: **SIMULATION HOMEWORKS**

Simulation homeworks come in the form of simulators you run to make sure you understand some piece of the material. The simulators are generally python programs that enable you both to *generate* different problems (using different random seeds) as well as to have the program solve the problem for you (with the `-c` flag) so that you can check your answers. Running any simulator with a `-h` or `--help` flag will provide with more information as to all the options the simulator gives you.

The README provided with each simulator gives more detail as to how to run it. Each flag is described in some detail therein.

This program, `process-run.py`, allows you to see how process states change as programs run and either use the CPU (e.g., perform an add instruction) or do I/O (e.g., send a request to a disk and wait for it to complete). See the README for details.

Questions

1. Run the program with the following flags: `./process-run.py -l 5:100,5:100`. What should the CPU utilization be (e.g., the percent of time the CPU is in use?) Why do you know this? Use the `-c` and `-p` flags to see if you were right.
2. Now run with these flags: `./process-run.py -l 4:100,1:0`. These flags specify one process with 4 instructions (all to use the CPU), and one that simply issues an I/O and waits for it to be done. How long does it take to complete both processes? Use `-c` and `-p` to find out if you were right.
3. Now switch the order of the processes: `./process-run.py -l 1:0,4:100`. What happens now? Does switching the order matter? Why? (As always, use `-c` and `-p` to see if you were right)
4. We'll now explore some of the other flags. One important flag is `-S`, which determines how the system reacts when a process issues an I/O. With the flag set to SWITCH_ON_END, the system will NOT switch to another process while one is doing I/O, instead waiting until the process is completely finished. What happens when you run the following two processes, one doing I/O and the other doing CPU work? (`-l 1:0,4:100 -c -S SWITCH_ON_END`)
5. Now, run the same processes, but with the switching behavior set to switch to another process whenever one is WAITING for I/O (`-l 1:0,4:100 -c -S SWITCH_ON_IO`). What happens now? Use `-c` and `-p` to confirm that you are right.
6. One other important behavior is what to do when an I/O completes. With `-I IO_RUN_LATER`, when an I/O completes, the pro-

cess that issued it is not necessarily run right away; rather, whatever was running at the time keeps running. What happens when you run this combination of processes? (`./process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -I IO_RUN_LATER -c -p`) Are system resources being effectively utilized?

7. Now run the same processes, but with `-I IO_RUN_IMMEDIATE` set, which immediately runs the process that issued the I/O. How does this behavior differ? Why might running a process that just completed an I/O again be a good idea?
8. Now run with some randomly generated processes, e.g., `-s 1 -l 3:50,3:50, -s 2 -l 3:50,3:50, -s 3 -l 3:50,3:50`. See if you can predict how the trace will turn out. What happens when you use `-I IO_RUN_IMMEDIATE` vs. `-I IO_RUN_LATER`? What happens when you use `-S SWITCH_ON_IO` vs. `-S SWITCH_ON_END`?

7

Scheduling: Introduction

By now low-level **mechanisms** of running processes (e.g., context switching) should be clear; if they are not, go back a chapter or two, and read the description of how that stuff works again. However, we have yet to understand the high-level **policies** that an OS scheduler employs. We will now do just that, presenting a series of **scheduling policies** (sometimes called **disciplines**) that various smart and hard-working people have developed over the years.

The origins of scheduling, in fact, predate computer systems; early approaches were taken from the field of operations management and applied to computers. This reality should be no surprise: assembly lines and many other human endeavors also require scheduling, and many of the same concerns exist therein, including a laser-like desire for efficiency. And thus, our problem:

THE CRUX: HOW TO DEVELOP SCHEDULING POLICY

How should we develop a basic framework for thinking about scheduling policies? What are the key assumptions? What metrics are important? What basic approaches have been used in the earliest of computer systems?

7.1 Workload Assumptions

Before getting into the range of possible policies, let us first make a number of simplifying assumptions about the processes running in the system, sometimes collectively called the **workload**. Determining the workload is a critical part of building policies, and the more you know about workload, the more fine-tuned your policy can be.

The workload assumptions we make here are mostly unrealistic, but that is alright (for now), because we will relax them as we go, and eventually develop what we will refer to as ... *(dramatic pause) ...*

a **fully-operational scheduling discipline**[1].

We will make the following assumptions about the processes, sometimes called **jobs**, that are running in the system:

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known.

We said many of these assumptions were unrealistic, but just as some animals are more equal than others in Orwell's *Animal Farm* [O45], some assumptions are more unrealistic than others in this chapter. In particular, it might bother you that the run-time of each job is known: this would make the scheduler omniscient, which, although it would be great (probably), is not likely to happen anytime soon.

7.2 Scheduling Metrics

Beyond making workload assumptions, we also need one more thing to enable us to compare different scheduling policies: a **scheduling metric**. A metric is just something that we use to *measure* something, and there are a number of different metrics that make sense in scheduling.

For now, however, let us also simplify our life by simply having a single metric: **turnaround time**. The turnaround time of a job is defined as the time at which the job completes minus the time at which the job arrived in the system. More formally, the turnaround time $T_{turnaround}$ is:

$$T_{turnaround} = T_{completion} - T_{arrival} \tag{7.1}$$

Because we have assumed that all jobs arrive at the same time, for now $T_{arrival} = 0$ and hence $T_{turnaround} = T_{completion}$. This fact will change as we relax the aforementioned assumptions.

You should note that turnaround time is a **performance** metric, which will be our primary focus this chapter. Another metric of interest is **fairness**, as measured (for example) by **Jain's Fairness Index** [J91]. Performance and fairness are often at odds in scheduling; a scheduler, for example, may optimize performance but at the cost of preventing a few jobs from running, thus decreasing fairness. This conundrum shows us that life isn't always perfect.

7.3 First In, First Out (FIFO)

The most basic algorithm we can implement is known as **First In, First Out** (**FIFO**) scheduling or sometimes **First Come, First Served** (**FCFS**).

[1] Said in the same way you would say "A fully-operational Death Star."

FIFO has a number of positive properties: it is clearly simple and thus easy to implement. And, given our assumptions, it works pretty well.

Let's do a quick example together. Imagine three jobs arrive in the system, A, B, and C, at roughly the same time ($T_{arrival} = 0$). Because FIFO has to put some job first, let's assume that while they all arrived simultaneously, A arrived just a hair before B which arrived just a hair before C. Assume also that each job runs for 10 seconds. What will the **average turnaround time** be for these jobs?

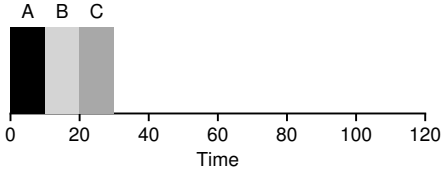


Figure 7.1: **FIFO Simple Example**

From Figure 7.1, you can see that A finished at 10, B at 20, and C at 30. Thus, the average turnaround time for the three jobs is simply $\frac{10+20+30}{3} = 20$. Computing turnaround time is as easy as that.

Now let's relax one of our assumptions. In particular, let's relax assumption 1, and thus no longer assume that each job runs for the same amount of time. How does FIFO perform now? What kind of workload could you construct to make FIFO perform poorly?

(think about this before reading on ... keep thinking ... got it?!)

Presumably you've figured this out by now, but just in case, let's do an example to show how jobs of different lengths can lead to trouble for FIFO scheduling. In particular, let's again assume three jobs (A, B, and C), but this time A runs for 100 seconds while B and C run for 10 each.

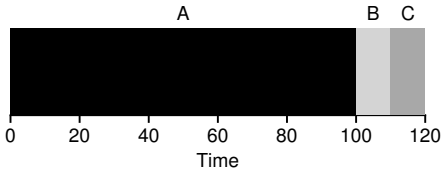


Figure 7.2: **Why FIFO Is Not That Great**

As you can see in Figure 7.2, Job A runs first for the full 100 seconds before B or C even get a chance to run. Thus, the average turnaround time for the system is high: a painful 110 seconds ($\frac{100+110+120}{3} = 110$).

This problem is generally referred to as the **convoy effect** [B+79], where a number of relatively-short potential consumers of a resource get queued behind a heavyweight resource consumer. This scheduling scenario might remind you of a single line at a grocery store and what you feel like when

TIP: THE PRINCIPLE OF SJF
Shortest Job First represents a general scheduling principle that can be applied to any system where the perceived turnaround time per customer (or, in our case, a job) matters. Think of any line you have waited in: if the establishment in question cares about customer satisfaction, it is likely they have taken SJF into account. For example, grocery stores commonly have a "ten-items-or-less" line to ensure that shoppers with only a few things to purchase don't get stuck behind the family preparing for some upcoming nuclear winter.

you see the person in front of you with three carts full of provisions and a checkbook out; it's going to be a while[2].

So what should we do? How can we develop a better algorithm to deal with our new reality of jobs that run for different amounts of time? Think about it first; then read on.

7.4 Shortest Job First (SJF)

It turns out that a very simple approach solves this problem; in fact it is an idea stolen from operations research [C54,PV56] and applied to scheduling of jobs in computer systems. This new scheduling discipline is known as **Shortest Job First (SJF)**, and the name should be easy to remember because it describes the policy quite completely: it runs the shortest job first, then the next shortest, and so on.

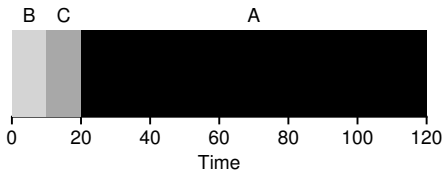


Figure 7.3: **SJF Simple Example**

Let's take our example above but with SJF as our scheduling policy. Figure 7.3 shows the results of running A, B, and C. Hopefully the diagram makes it clear why SJF performs much better with regards to average turnaround time. Simply by running B and C before A, SJF reduces average turnaround from 110 seconds to 50 ($\frac{10+20+120}{3} = 50$), more than a factor of two improvement.

In fact, given our assumptions about jobs all arriving at the same time, we could prove that SJF is indeed an **optimal** scheduling algorithm. How-

[2]Recommended action in this case: either quickly switch to a different line, or take a long, deep, and relaxing breath. That's right, breathe in, breathe out. It will be OK, don't worry.

ASIDE: **PREEMPTIVE SCHEDULERS**
In the old days of batch computing, a number of **non-preemptive** schedulers were developed; such systems would run each job to completion before considering whether to run a new job. Virtually all modern schedulers are **preemptive**, and quite willing to stop one process from running in order to run another. This implies that the scheduler employs the mechanisms we learned about previously; in particular, the scheduler can perform a **context switch**, stopping one running process temporarily and resuming (or starting) another.

ever, you are in a systems class, not theory or operations research; no proofs are allowed.

Thus we arrive upon a good approach to scheduling with SJF, but our assumptions are still fairly unrealistic. Let's relax another. In particular, we can target assumption 2, and now assume that jobs can arrive at any time instead of all at once. What problems does this lead to?

(Another pause to think ... are you thinking? Come on, you can do it)

Here we can illustrate the problem again with an example. This time, assume A arrives at $t = 0$ and needs to run for 100 seconds, whereas B and C arrive at $t = 10$ and each need to run for 10 seconds. With pure SJF, we'd get the schedule seen in Figure 7.4.

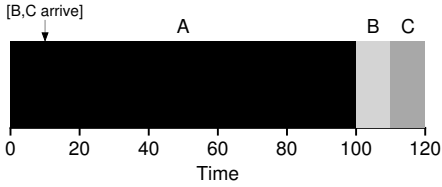


Figure 7.4: **SJF With Late Arrivals From B and C**

As you can see from the figure, even though B and C arrived shortly after A, they still are forced to wait until A has completed, and thus suffer the same convoy problem. Average turnaround time for these three jobs is 103.33 seconds ($\frac{100+(110-10)+(120-10)}{3}$). What can a scheduler do?

7.5 Shortest Time-to-Completion First (STCF)

To address this concern, we need to relax assumption 3 (that jobs must run to completion), so let's do that. We also need some machinery within the scheduler itself. As you might have guessed, given our previous discussion about timer interrupts and context switching, the scheduler can certainly do something else when B and C arrive: it can **preempt** job A and decide to run another job, perhaps continuing A later. SJF by our definition is a **non-preemptive** scheduler, and thus suffers from the problems described above.

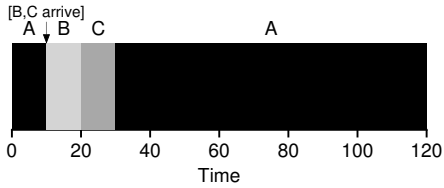


Figure 7.5: **STCF Simple Example**

Fortunately, there is a scheduler which does exactly that: add preemption to SJF, known as the **Shortest Time-to-Completion First** (**STCF**) or **Preemptive Shortest Job First** (**PSJF**) scheduler [CK68]. Any time a new job enters the system, the STCF scheduler determines which of the remaining jobs (including the new job) has the least time left, and schedules that one. Thus, in our example, STCF would preempt A and run B and C to completion; only when they are finished would A's remaining time be scheduled. Figure 7.5 shows an example.

The result is a much-improved average turnaround time: 50 seconds ($\frac{(120-0)+(20-10)+(30-10)}{3}$). And as before, given our new assumptions, STCF is provably optimal; given that SJF is optimal if all jobs arrive at the same time, you should probably be able to see the intuition behind the optimality of STCF.

7.6 A New Metric: Response Time

Thus, if we knew job lengths, and that jobs only used the CPU, and our only metric was turnaround time, STCF would be a great policy. In fact, for a number of early batch computing systems, these types of scheduling algorithms made some sense. However, the introduction of time-shared machines changed all that. Now users would sit at a terminal and demand interactive performance from the system as well. And thus, a new metric was born: **response time**.

Response time is defined as the time from when the job arrives in a system to the first time it is scheduled. More formally:

$$T_{response} = T_{firstrun} - T_{arrival} \tag{7.2}$$

For example, if we had the schedule above (with A arriving at time 0, and B and C at time 10), the response time of each job is as follows: 0 for job A, 0 for B, and 10 for C (average: 3.33).

As you might be thinking, STCF and related disciplines are not particularly good for response time. If three jobs arrive at the same time, for example, the third job has to wait for the previous two jobs to run *in their entirety* before being scheduled just once. While great for turnaround time, this approach is quite bad for response time and interactivity. Indeed, imagine sitting at a terminal, typing, and having to wait 10 seconds

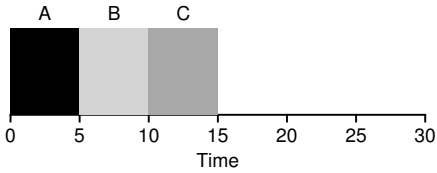


Figure 7.6: **SJF Again (Bad for Response Time)**

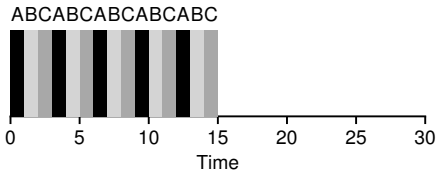


Figure 7.7: **Round Robin (Good for Response Time)**

to see a response from the system just because some other job got scheduled in front of yours: not too pleasant.

Thus, we are left with another problem: how can we build a scheduler that is sensitive to response time?

7.7 Round Robin

To solve this problem, we will introduce a new scheduling algorithm, classically referred to as **Round-Robin (RR)** scheduling [K64]. The basic idea is simple: instead of running jobs to completion, RR runs a job for a **time slice** (sometimes called a **scheduling quantum**) and then switches to the next job in the run queue. It repeatedly does so until the jobs are finished. For this reason, RR is sometimes called **time-slicing**. Note that the length of a time slice must be a multiple of the timer-interrupt period; thus if the timer interrupts every 10 milliseconds, the time slice could be 10, 20, or any other multiple of 10 ms.

To understand RR in more detail, let's look at an example. Assume three jobs A, B, and C arrive at the same time in the system, and that they each wish to run for 5 seconds. An SJF scheduler runs each job to completion before running another (Figure 7.6). In contrast, RR with a time-slice of 1 second would cycle through the jobs quickly (Figure 7.7).

The average response time of RR is: $\frac{0+1+2}{3} = 1$; for SJF, average response time is: $\frac{0+5+10}{3} = 5$.

As you can see, the length of the time slice is critical for RR. The shorter it is, the better the performance of RR under the response-time metric. However, making the time slice too short is problematic: suddenly the cost of context switching will dominate overall performance. Thus, deciding on the length of the time slice presents a trade-off to a system designer, making it long enough to **amortize** the cost of switching without making it so long that the system is no longer responsive.

TIP: AMORTIZATION CAN REDUCE COSTS
The general technique of **amortization** is commonly used in systems when there is a fixed cost to some operation. By incurring that cost less often (i.e., by performing the operation fewer times), the total cost to the system is reduced. For example, if the time slice is set to 10 ms, and the context-switch cost is 1 ms, roughly 10% of time is spent context switching and is thus wasted. If we want to *amortize* this cost, we can increase the time slice, e.g., to 100 ms. In this case, less than 1% of time is spent context switching, and thus the cost of time-slicing has been amortized.

Note that the cost of context switching does not arise solely from the OS actions of saving and restoring a few registers. When programs run, they build up a great deal of state in CPU caches, TLBs, branch predictors, and other on-chip hardware. Switching to another job causes this state to be flushed and new state relevant to the currently-running job to be brought in, which may exact a noticeable performance cost [MB91].

RR, with a reasonable time slice, is thus an excellent scheduler if response time is our only metric. But what about our old friend turnaround time? Let's look at our example above again. A, B, and C, each with running times of 5 seconds, arrive at the same time, and RR is the scheduler with a (long) 1-second time slice. We can see from the picture above that A finishes at 13, B at 14, and C at 15, for an average of 14. Pretty awful!

It is not surprising, then, that RR is indeed one of the *worst* policies if turnaround time is our metric. Intuitively, this should make sense: what RR is doing is stretching out each job as long as it can, by only running each job for a short bit before moving to the next. Because turnaround time only cares about when jobs finish, RR is nearly pessimal, even worse than simple FIFO in many cases.

More generally, any policy (such as RR) that is **fair**, i.e., that evenly divides the CPU among active processes on a small time scale, will perform poorly on metrics such as turnaround time. Indeed, this is an inherent trade-off: if you are willing to be unfair, you can run shorter jobs to completion, but at the cost of response time; if you instead value fairness, response time is lowered, but at the cost of turnaround time. This type of **trade-off** is common in systems; you can't have your cake and eat it too[3].

We have developed two types of schedulers. The first type (SJF, STCF) optimizes turnaround time, but is bad for response time. The second type (RR) optimizes response time but is bad for turnaround. And we still have two assumptions which need to be relaxed: assumption 4 (that jobs do no I/O), and assumption 5 (that the run-time of each job is known). Let's tackle those assumptions next.

[3] A saying that confuses people, because it should be "You can't *keep* your cake and eat it too" (which is kind of obvious, no?). Amazingly, there is a wikipedia page about this saying; even more amazingly, it is kind of fun to read [W15]. As they say in Italian, you can't *Avere la botte piena e la moglie ubriaca.*

TIP: OVERLAP ENABLES HIGHER UTILIZATION
When possible, **overlap** operations to maximize the utilization of systems. Overlap is useful in many different domains, including when performing disk I/O or sending messages to remote machines; in either case, starting the operation and then switching to other work is a good idea, and improves the overall utilization and efficiency of the system.

7.8 Incorporating I/O

First we will relax assumption 4 — of course all programs perform I/O. Imagine a program that didn't take any input: it would produce the same output each time. Imagine one without output: it is the proverbial tree falling in the forest, with no one to see it; it doesn't matter that it ran.

A scheduler clearly has a decision to make when a job initiates an I/O request, because the currently-running job won't be using the CPU during the I/O; it is **blocked** waiting for I/O completion. If the I/O is sent to a hard disk drive, the process might be blocked for a few milliseconds or longer, depending on the current I/O load of the drive. Thus, the scheduler should probably schedule another job on the CPU at that time.

The scheduler also has to make a decision when the I/O completes. When that occurs, an interrupt is raised, and the OS runs and moves the process that issued the I/O from blocked back to the ready state. Of course, it could even decide to run the job at that point. How should the OS treat each job?

To understand this issue better, let us assume we have two jobs, A and B, which each need 50 ms of CPU time. However, there is one obvious difference: A runs for 10 ms and then issues an I/O request (assume here that I/Os each take 10 ms), whereas B simply uses the CPU for 50 ms and performs no I/O. The scheduler runs A first, then B after (Figure 7.8).

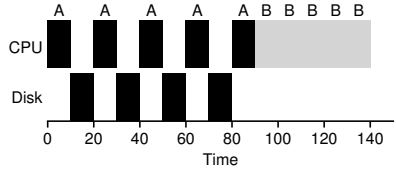


Figure 7.8: **Poor Use of Resources**

Assume we are trying to build a STCF scheduler. How should such a scheduler account for the fact that A is broken up into 5 10-ms sub-jobs, whereas B is just a single 50-ms CPU demand? Clearly, just running one job and then the other without considering how to take I/O into account makes little sense.

A common approach is to treat each 10-ms sub-job of A as an independent job. Thus, when the system starts, its choice is whether to schedule

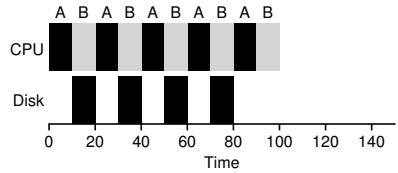


Figure 7.9: **Overlap Allows Better Use of Resources**

a 10-ms A or a 50-ms B. With STCF, the choice is clear: choose the shorter one, in this case A. Then, when the first sub-job of A has completed, only B is left, and it begins running. Then a new sub-job of A is submitted, and it preempts B and runs for 10 ms. Doing so allows for **overlap**, with the CPU being used by one process while waiting for the I/O of another process to complete; the system is thus better utilized (see Figure 7.9).

And thus we see how a scheduler might incorporate I/O. By treating each CPU burst as a job, the scheduler makes sure processes that are "interactive" get run frequently. While those interactive jobs are performing I/O, other CPU-intensive jobs run, thus better utilizing the processor.

7.9 No More Oracle

With a basic approach to I/O in place, we come to our final assumption: that the scheduler knows the length of each job. As we said before, this is likely the worst assumption we could make. In fact, in a general-purpose OS (like the ones we care about), the OS usually knows very little about the length of each job. Thus, how can we build an approach that behaves like SJF/STCF without such *a priori* knowledge? Further, how can we incorporate some of the ideas we have seen with the RR scheduler so that response time is also quite good?

7.10 Summary

We have introduced the basic ideas behind scheduling and developed two families of approaches. The first runs the shortest job remaining and thus optimizes turnaround time; the second alternates between all jobs and thus optimizes response time. Both are bad where the other is good, alas, an inherent trade-off common in systems. We have also seen how we might incorporate I/O into the picture, but have still not solved the problem of the fundamental inability of the OS to see into the future. Shortly, we will see how to overcome this problem, by building a scheduler that uses the recent past to predict the future. This scheduler is known as the **multi-level feedback queue**, and it is the topic of the next chapter.

References

[B+79] "The Convoy Phenomenon"
M. Blasgen, J. Gray, M. Mitoma, T. Price
ACM Operating Systems Review, 13:2, April 1979
Perhaps the first reference to convoys, which occurs in databases as well as the OS.

[C54] "Priority Assignment in Waiting Line Problems"
A. Cobham
Journal of Operations Research, 2:70, pages 70–76, 1954
The pioneering paper on using an SJF approach in scheduling the repair of machines.

[K64] "Analysis of a Time-Shared Processor"
Leonard Kleinrock
Naval Research Logistics Quarterly, 11:1, pages 59–73, March 1964
May be the first reference to the round-robin scheduling algorithm; certainly one of the first analyses of said approach to scheduling a time-shared system.

[CK68] "Computer Scheduling Methods and their Countermeasures"
Edward G. Coffman and Leonard Kleinrock
AFIPS '68 (Spring), April 1968
An excellent early introduction to and analysis of a number of basic scheduling disciplines.

[J91] "The Art of Computer Systems Performance Analysis:
Techniques for Experimental Design, Measurement, Simulation, and Modeling"
R. Jain
Interscience, New York, April 1991
The standard text on computer systems measurement. A great reference for your library, for sure.

[O45] "Animal Farm"
George Orwell
Secker and Warburg (London), 1945
A great but depressing allegorical book about power and its corruptions. Some say it is a critique of Stalin and the pre-WWII Stalin era in the U.S.S.R; we say it's a critique of pigs.

[PV56] "Machine Repair as a Priority Waiting-Line Problem"
Thomas E. Phipps Jr. and W. R. Van Voorhis
Operations Research, 4:1, pages 76–86, February 1956
Follow-on work that generalizes the SJF approach to machine repair from Cobham's original work; also postulates the utility of an STCF approach in such an environment. Specifically, "There are certain types of repair work, ... involving much dismantling and covering the floor with nuts and bolts, which certainly should not be interrupted once undertaken; in other cases it would be inadvisable to continue work on a long job if one or more short ones became available (p.81)."

[MB91] "The effect of context switches on cache performance"
Jeffrey C. Mogul and Anita Borg
ASPLOS, 1991
A nice study on how cache performance can be affected by context switching; less of an issue in today's systems where processors issue billions of instructions per second but context-switches still happen in the millisecond time range.

[W15] "You can't have your cake and eat it"
`http://en.wikipedia.org/wiki/You_can't_have_your_cake_and_eat_it`
Wikipedia, as of December 2015
The best part of this page is reading all the similar idioms from other languages. In Tamil, you can't "have both the moustache and drink the soup."

Homework

This program, `scheduler.py`, allows you to see how different schedulers perform under scheduling metrics such as response time, turnaround time, and total wait time. See the README for details.

Questions

1. Compute the response time and turnaround time when running three jobs of length 200 with the SJF and FIFO schedulers.
2. Now do the same but with jobs of different lengths: 100, 200, and 300.
3. Now do the same, but also with the RR scheduler and a time-slice of 1.
4. For what types of workloads does SJF deliver the same turnaround times as FIFO?
5. For what types of workloads and quantum lengths does SJF deliver the same response times as RR?
6. What happens to response time with SJF as job lengths increase? Can you use the simulator to demonstrate the trend?
7. What happens to response time with RR as quantum lengths increase? Can you write an equation that gives the worst-case response time, given N jobs?

6

Mechanism: Limited Direct Execution

In order to virtualize the CPU, the operating system needs to somehow share the physical CPU among many jobs running seemingly at the same time. The basic idea is simple: run one process for a little while, then run another one, and so forth. By **time sharing** the CPU in this manner, virtualization is achieved.

There are a few challenges, however, in building such virtualization machinery. The first is *performance*: how can we implement virtualization without adding excessive overhead to the system? The second is *control*: how can we run processes efficiently while retaining control over the CPU? Control is particularly important to the OS, as it is in charge of resources; without control, a process could simply run forever and take over the machine, or access information that it should not be allowed to access. Obtaining high performance while maintaining control is thus one of the central challenges in building an operating system.

THE CRUX:
HOW TO EFFICIENTLY VIRTUALIZE THE CPU WITH CONTROL

The OS must virtualize the CPU in an efficient manner while retaining control over the system. To do so, both hardware and operating-system support will be required. The OS will often use a judicious bit of hardware support in order to accomplish its work effectively.

6.1 Basic Technique: Limited Direct Execution

To make a program run as fast as one might expect, not surprisingly OS developers came up with a technique, which we call **limited direct execution**. The "direct execution" part of the idea is simple: just run the program directly on the CPU. Thus, when the OS wishes to start a program running, it creates a process entry for it in a process list, allocates some memory for it, loads the program code into memory (from disk), locates its entry point (i.e., the `main()` routine or something similar), jumps

OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute **call** main()	
	Run main()
	Execute **return** from main
Free memory of process	
Remove from process list	

Figure 6.1: **Direct Execution Protocol (Without Limits)**

to it, and starts running the user's code. Figure 6.1 shows this basic direct execution protocol (without any limits, yet), using a normal call and return to jump to the program's `main()` and later to get back into the kernel.

Sounds simple, no? But this approach gives rise to a few problems in our quest to virtualize the CPU. The first is simple: if we just run a program, how can the OS make sure the program doesn't do anything that we don't want it to do, while still running it efficiently? The second: when we are running a process, how does the operating system stop it from running and switch to another process, thus implementing the **time sharing** we require to virtualize the CPU?

In answering these questions below, we'll get a much better sense of what is needed to virtualize the CPU. In developing these techniques, we'll also see where the "limited" part of the name arises from; without limits on running programs, the OS wouldn't be in control of anything and thus would be "just a library" — a very sad state of affairs for an aspiring operating system!

6.2 Problem #1: Restricted Operations

Direct execution has the obvious advantage of being fast; the program runs natively on the hardware CPU and thus executes as quickly as one would expect. But running on the CPU introduces a problem: what if the process wishes to perform some kind of restricted operation, such as issuing an I/O request to a disk, or gaining access to more system resources such as CPU or memory?

THE CRUX: HOW TO PERFORM RESTRICTED OPERATIONS

A process must be able to perform I/O and some other restricted operations, but without giving the process complete control over the system. How can the OS and hardware work together to do so?

ASIDE: **WHY SYSTEM CALLS LOOK LIKE PROCEDURE CALLS**

You may wonder why a call to a system call, such as `open()` or `read()`, looks exactly like a typical procedure call in C; that is, if it looks just like a procedure call, how does the system know it's a system call, and do all the right stuff? The simple reason: it *is* a procedure call, but hidden inside that procedure call is the famous trap instruction. More specifically, when you call `open()` (for example), you are executing a procedure call into the C library. Therein, whether for `open()` or any of the other system calls provided, the library uses an agreed-upon calling convention with the kernel to put the arguments to open in well-known locations (e.g., on the stack, or in specific registers), puts the system-call number into a well-known location as well (again, onto the stack or a register), and then executes the aforementioned trap instruction. The code in the library after the trap unpacks return values and returns control to the program that issued the system call. Thus, the parts of the C library that make system calls are hand-coded in assembly, as they need to carefully follow convention in order to process arguments and return values correctly, as well as execute the hardware-specific trap instruction. And now you know why you personally don't have to write assembly code to trap into an OS; somebody has already written that assembly for you.

One approach would simply be to let any process do whatever it wants in terms of I/O and other related operations. However, doing so would prevent the construction of many kinds of systems that are desirable. For example, if we wish to build a file system that checks permissions before granting access to a file, we can't simply let any user process issue I/Os to the disk; if we did, a process could simply read or write the entire disk and thus all protections would be lost.

Thus, the approach we take is to introduce a new processor mode, known as **user mode**; code that runs in user mode is restricted in what it can do. For example, when running in user mode, a process can't issue I/O requests; doing so would result in the processor raising an exception; the OS would then likely kill the process.

In contrast to user mode is **kernel mode**, which the operating system (or kernel) runs in. In this mode, code that runs can do what it likes, including privileged operations such as issuing I/O requests and executing all types of restricted instructions.

We are still left with a challenge, however: what should a user process do when it wishes to perform some kind of privileged operation, such as reading from disk? To enable this, virtually all modern hardware provides the ability for user programs to perform a **system call**. Pioneered on ancient machines such as the Atlas [K+61,L78], system calls allow the kernel to carefully expose certain key pieces of functionality to user programs, such as accessing the file system, creating and destroying processes, communicating with other processes, and allocating more

TIP: USE PROTECTED CONTROL TRANSFER
The hardware assists the OS by providing different modes of execution. In **user mode**, applications do not have full access to hardware resources. In **kernel mode**, the OS has access to the full resources of the machine. Special instructions to **trap** into the kernel and **return-from-trap** back to user-mode programs are also provided, as well as instructions that allow the OS to tell the hardware where the **trap table** resides in memory.

memory. Most operating systems provide a few hundred calls (see the POSIX standard for details [P10]); early Unix systems exposed a more concise subset of around twenty calls.

To execute a system call, a program must execute a special **trap** instruction. This instruction simultaneously jumps into the kernel and raises the privilege level to kernel mode; once in the kernel, the system can now perform whatever privileged operations are needed (if allowed), and thus do the required work for the calling process. When finished, the OS calls a special **return-from-trap** instruction, which, as you might expect, returns into the calling user program while simultaneously reducing the privilege level back to user mode.

The hardware needs to be a bit careful when executing a trap, in that it must make sure to save enough of the caller's registers in order to be able to return correctly when the OS issues the return-from-trap instruction. On x86, for example, the processor will push the program counter, flags, and a few other registers onto a per-process **kernel stack**; the return-from-trap will pop these values off the stack and resume execution of the user-mode program (see the Intel systems manuals [I11] for details). Other hardware systems use different conventions, but the basic concepts are similar across platforms.

There is one important detail left out of this discussion: how does the trap know which code to run inside the OS? Clearly, the calling process can't specify an address to jump to (as you would when making a procedure call); doing so would allow programs to jump anywhere into the kernel which clearly is a **Very Bad Idea**[1]. Thus the kernel must carefully control what code executes upon a trap.

The kernel does so by setting up a **trap table** at boot time. When the machine boots up, it does so in privileged (kernel) mode, and thus is free to configure machine hardware as need be. One of the first things the OS thus does is to tell the hardware what code to run when certain exceptional events occur. For example, what code should run when a hard-disk interrupt takes place, when a keyboard interrupt occurs, or when a program makes a system call? The OS informs the hardware of the locations of these **trap handlers**, usually with some kind of special in-

[1] Imagine jumping into code to access a file, but just after a permission check; in fact, it is likely such an ability would enable a wily programmer to get the kernel to run arbitrary code sequences [S07]. In general, try to avoid Very Bad Ideas like this one.

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember address of... syscall handler	

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list		
Allocate memory for program		
Load program into memory		
Setup user stack with argv		
Fill kernel stack with reg/PC		
return-from-trap		
	restore regs from kernel stack	
	move to user mode	
	jump to main	
		Run main()
		...
		Call system call **trap** into OS
	save regs to kernel stack	
	move to kernel mode	
	jump to trap handler	
Handle trap		
Do work of syscall		
return-from-trap		
	restore regs from kernel stack	
	move to user mode	
	jump to PC after trap	
		...
		return from main **trap** (via `exit()`)
Free memory of process		
Remove from process list		

Figure 6.2: **Limited Direct Execution Protocol**

struction. Once the hardware is informed, it remembers the location of these handlers until the machine is next rebooted, and thus the hardware knows what to do (i.e., what code to jump to) when system calls and other exceptional events take place.

To specify the exact system call, a **system-call number** is usually assigned to each system call. The user code is thus responsible for placing the desired system-call number in a register or at a specified location on the stack; the OS, when handling the system call inside the trap handler, examines this number, ensures it is valid, and, if it is, executes the corresponding code. This level of indirection serves as a form of **protection**; user code cannot specify an exact address to jump to, but rather must request a particular service via number.

One last aside: being able to execute the instruction to tell the hardware where the trap tables are is a very powerful capability. Thus, as you might have guessed, it is also a **privileged** operation. If you try to execute this instruction in user mode, the hardware won't let you, and you

can probably guess what will happen (hint: adios, offending program). Point to ponder: what horrible things could you do to a system if you could install your own trap table? Could you take over the machine?

The timeline (with time increasing downward, in Figure 6.2) summarizes the protocol. We assume each process has a kernel stack where registers (including general purpose registers and the program counter) are saved to and restored from (by the hardware) when transitioning into and out of the kernel.

There are two phases in the LDE protocol. In the first (at boot time), the kernel initializes the trap table, and the CPU remembers its location for subsequent use. The kernel does so via a privileged instruction (all privileged instructions are highlighted in bold).

In the second (when running a process), the kernel sets up a few things (e.g., allocating a node on the process list, allocating memory) before using a return-from-trap instruction to start the execution of the process; this switches the CPU to user mode and begins running the process. When the process wishes to issue a system call, it traps back into the OS, which handles it and once again returns control via a return-from-trap to the process. The process then completes its work, and returns from `main()`; this usually will return into some stub code which will properly exit the program (say, by calling the `exit()` system call, which traps into the OS). At this point, the OS cleans up and we are done.

6.3 Problem #2: Switching Between Processes

The next problem with direct execution is achieving a switch between processes. Switching between processes should be simple, right? The OS should just decide to stop one process and start another. What's the big deal? But it actually is a little bit tricky: specifically, if a process is running on the CPU, this by definition means the OS is *not* running. If the OS is not running, how can it do anything at all? (hint: it can't) While this sounds almost philosophical, it is a real problem: there is clearly no way for the OS to take an action if it is not running on the CPU. Thus we arrive at the crux of the problem.

THE CRUX: HOW TO REGAIN CONTROL OF THE CPU
How can the operating system **regain control** of the CPU so that it can switch between processes?

A Cooperative Approach: Wait For System Calls

One approach that some systems have taken in the past (for example, early versions of the Macintosh operating system [M11], or the old Xerox Alto system [A79]) is known as the **cooperative** approach. In this style,

TIP: DEALING WITH APPLICATION MISBEHAVIOR
Operating systems often have to deal with misbehaving processes, those that either through design (maliciousness) or accident (bugs) attempt to do something that they shouldn't. In modern systems, the way the OS tries to handle such malfeasance is to simply terminate the offender. One strike and you're out! Perhaps brutal, but what else should the OS do when you try to access memory illegally or execute an illegal instruction?

the OS *trusts* the processes of the system to behave reasonably. Processes that run for too long are assumed to periodically give up the CPU so that the OS can decide to run some other task.

Thus, you might ask, how does a friendly process give up the CPU in this utopian world? Most processes, as it turns out, transfer control of the CPU to the OS quite frequently by making **system calls**, for example, to open a file and subsequently read it, or to send a message to another machine, or to create a new process. Systems like this often include an explicit **yield** system call, which does nothing except to transfer control to the OS so it can run other processes.

Applications also transfer control to the OS when they do something illegal. For example, if an application divides by zero, or tries to access memory that it shouldn't be able to access, it will generate a **trap** to the OS. The OS will then have control of the CPU again (and likely terminate the offending process).

Thus, in a cooperative scheduling system, the OS regains control of the CPU by waiting for a system call or an illegal operation of some kind to take place. You might also be thinking: isn't this passive approach less than ideal? What happens, for example, if a process (whether malicious, or just full of bugs) ends up in an infinite loop, and never makes a system call? What can the OS do then?

A Non-Cooperative Approach: The OS Takes Control

Without some additional help from the hardware, it turns out the OS can't do much at all when a process refuses to make system calls (or mistakes) and thus return control to the OS. In fact, in the cooperative approach, your only recourse when a process gets stuck in an infinite loop is to resort to the age-old solution to all problems in computer systems: **reboot the machine**. Thus, we again arrive at a subproblem of our general quest to gain control of the CPU.

THE CRUX: HOW TO GAIN CONTROL WITHOUT COOPERATION
How can the OS gain control of the CPU even if processes are not being cooperative? What can the OS do to ensure a rogue process does not take over the machine?

TIP: USE THE TIMER INTERRUPT TO REGAIN CONTROL
The addition of a **timer interrupt** gives the OS the ability to run again on a CPU even if processes act in a non-cooperative fashion. Thus, this hardware feature is essential in helping the OS maintain control of the machine.

The answer turns out to be simple and was discovered by a number of people building computer systems many years ago: a **timer interrupt** [M+63]. A timer device can be programmed to raise an interrupt every so many milliseconds; when the interrupt is raised, the currently running process is halted, and a pre-configured **interrupt handler** in the OS runs. At this point, the OS has regained control of the CPU, and thus can do what it pleases: stop the current process, and start a different one.

As we discussed before with system calls, the OS must inform the hardware of which code to run when the timer interrupt occurs; thus, at boot time, the OS does exactly that. Second, also during the boot sequence, the OS must start the timer, which is of course a privileged operation. Once the timer has begun, the OS can thus feel safe in that control will eventually be returned to it, and thus the OS is free to run user programs. The timer can also be turned off (also a privileged operation), something we will discuss later when we understand concurrency in more detail.

Note that the hardware has some responsibility when an interrupt occurs, in particular to save enough of the state of the program that was running when the interrupt occurred such that a subsequent return-from-trap instruction will be able to resume the running program correctly. This set of actions is quite similar to the behavior of the hardware during an explicit system-call trap into the kernel, with various registers thus getting saved (e.g., onto a kernel stack) and thus easily restored by the return-from-trap instruction.

Saving and Restoring Context

Now that the OS has regained control, whether cooperatively via a system call, or more forcefully via a timer interrupt, a decision has to be made: whether to continue running the currently-running process, or switch to a different one. This decision is made by a part of the operating system known as the **scheduler**; we will discuss scheduling policies in great detail in the next few chapters.

If the decision is made to switch, the OS then executes a low-level piece of code which we refer to as a **context switch**. A context switch is conceptually simple: all the OS has to do is save a few register values for the currently-executing process (onto its kernel stack, for example) and restore a few for the soon-to-be-executing process (from its kernel stack). By doing so, the OS thus ensures that when the return-from-trap

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember addresses of...
	syscall handler
	timer handler
start interrupt timer	
	start timer
	interrupt CPU in X ms

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	timer interrupt	
	save regs(A) to k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) to proc-struct(A)		
restore regs(B) from proc-struct(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) from k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

Figure 6.3: **Limited Direct Execution Protocol (Timer Interrupt)**

instruction is finally executed, instead of returning to the process that was running, the system resumes execution of another process.

To save the context of the currently-running process, the OS will execute some low-level assembly code to save the general purpose registers, PC, as well as the kernel stack pointer of the currently-running process, and then restore said registers, PC, and switch to the kernel stack for the soon-to-be-executing process. By switching stacks, the kernel enters the call to the switch code in the context of one process (the one that was interrupted) and returns in the context of another (the soon-to-be-executing one). When the OS then finally executes a return-from-trap instruction, the soon-to-be-executing process becomes the currently-running process. And thus the context switch is complete.

A timeline of the entire process is shown in Figure 6.3. In this example, Process A is running and then is interrupted by the timer interrupt. The hardware saves its registers (onto its kernel stack) and enters the kernel (switching to kernel mode). In the timer interrupt handler, the OS decides to switch from running Process A to Process B. At that point, it calls the `switch()` routine, which carefully saves current register values (into the process structure of A), restores the registers of Process B (from its process structure entry), and then **switches contexts**, specifically by changing the

```
1  # void swtch(struct context **old, struct context *new);
2  #
3  # Save current register context in old
4  # and then load register context from new.
5  .globl swtch
6  swtch:
7    # Save old registers
8    movl 4(%esp), %eax  # put old ptr into eax
9    popl 0(%eax)        # save the old IP
10   movl %esp, 4(%eax)  # and stack
11   movl %ebx, 8(%eax)  # and other registers
12   movl %ecx, 12(%eax)
13   movl %edx, 16(%eax)
14   movl %esi, 20(%eax)
15   movl %edi, 24(%eax)
16   movl %ebp, 28(%eax)
17
18   # Load new registers
19   movl 4(%esp), %eax  # put new ptr into eax
20   movl 28(%eax), %ebp # restore other registers
21   movl 24(%eax), %edi
22   movl 20(%eax), %esi
23   movl 16(%eax), %edx
24   movl 12(%eax), %ecx
25   movl 8(%eax), %ebx
26   movl 4(%eax), %esp  # stack is switched here
27   pushl 0(%eax)       # return addr put in place
28   ret                 # finally return into new ctxt
```

Figure 6.4: **The xv6 Context Switch Code**

stack pointer to use B's kernel stack (and not A's). Finally, the OS returns-from-trap, which restores B's registers and starts running it.

Note that there are two types of register saves/restores that happen during this protocol. The first is when the timer interrupt occurs; in this case, the *user registers* of the running process are implicitly saved by the *hardware*, using the kernel stack of that process. The second is when the OS decides to switch from A to B; in this case, the *kernel registers* are explicitly saved by the *software* (i.e., the OS), but this time into memory in the process structure of the process. The latter action moves the system from running as if it just trapped into the kernel from A to as if it just trapped into the kernel from B.

To give you a better sense of how such a switch is enacted, Figure 6.4 shows the context switch code for xv6. See if you can make sense of it (you'll have to know a bit of x86, as well as some xv6, to do so). The `context` structures `old` and `new` are found in the old and new process's process structures, respectively.

6.4 Worried About Concurrency?

Some of you, as attentive and thoughtful readers, may be now thinking: "Hmm... what happens when, during a system call, a timer interrupt

ASIDE: **HOW LONG CONTEXT SWITCHES TAKE**

A natural question you might have is: how long does something like a context switch take? Or even a system call? For those of you that are curious, there is a tool called **lmbench** [MS96] that measures exactly those things, as well as a few other performance measures that might be relevant.

Results have improved quite a bit over time, roughly tracking processor performance. For example, in 1996 running Linux 1.3.37 on a 200-MHz P6 CPU, system calls took roughly 4 microseconds, and a context switch roughly 6 microseconds [MS96]. Modern systems perform almost an order of magnitude better, with sub-microsecond results on systems with 2- or 3-GHz processors.

It should be noted that not all operating-system actions track CPU performance. As Ousterhout observed, many OS operations are memory intensive, and memory bandwidth has not improved as dramatically as processor speed over time [O90]. Thus, depending on your workload, buying the latest and greatest processor may not speed up your OS as much as you might hope.

occurs?" or "What happens when you're handling one interrupt and another one happens? Doesn't that get hard to handle in the kernel?" Good questions — we really have some hope for you yet!

The answer is yes, the OS does indeed need to be concerned as to what happens if, during interrupt or trap handling, another interrupt occurs. This, in fact, is the exact topic of the entire second piece of this book, on **concurrency**; we'll defer a detailed discussion until then.

To whet your appetite, we'll just sketch some basics of how the OS handles these tricky situations. One simple thing an OS might do is **disable interrupts** during interrupt processing; doing so ensures that when one interrupt is being handled, no other one will be delivered to the CPU. Of course, the OS has to be careful in doing so; disabling interrupts for too long could lead to lost interrupts, which is (in technical terms) bad.

Operating systems also have developed a number of sophisticated **locking** schemes to protect concurrent access to internal data structures. This enables multiple activities to be on-going within the kernel at the same time, particularly useful on multiprocessors. As we'll see in the next piece of this book on concurrency, though, such locking can be complicated and lead to a variety of interesting and hard-to-find bugs.

6.5 Summary

We have described some key low-level mechanisms to implement CPU virtualization, a set of techniques which we collectively refer to as **limited direct execution**. The basic idea is straightforward: just run the program you want to run on the CPU, but first make sure to set up the hardware so as to limit what the process can do without OS assistance.

TIP: REBOOT IS USEFUL

Earlier on, we noted that the only solution to infinite loops (and similar behaviors) under cooperative preemption is to **reboot** the machine. While you may scoff at this hack, researchers have shown that reboot (or in general, starting over some piece of software) can be a hugely useful tool in building robust systems [C+04].

Specifically, reboot is useful because it moves software back to a known and likely more tested state. Reboots also reclaim stale or leaked resources (e.g., memory) which may otherwise be hard to handle. Finally, reboots are easy to automate. For all of these reasons, it is not uncommon in large-scale cluster Internet services for system management software to periodically reboot sets of machines in order to reset them and thus obtain the advantages listed above.

Thus, next time you reboot, you are not just enacting some ugly hack. Rather, you are using a time-tested approach to improving the behavior of a computer system. Well done!

This general approach is taken in real life as well. For example, those of you who have children, or, at least, have heard of children, may be familiar with the concept of **baby proofing** a room: locking cabinets containing dangerous stuff and covering electrical sockets. When the room is thus readied, you can let your baby roam freely, secure in the knowledge that the most dangerous aspects of the room have been restricted.

In an analogous manner, the OS "baby proofs" the CPU, by first (during boot time) setting up the trap handlers and starting an interrupt timer, and then by only running processes in a restricted mode. By doing so, the OS can feel quite assured that processes can run efficiently, only requiring OS intervention to perform privileged operations or when they have monopolized the CPU for too long and thus need to be switched out.

We thus have the basic mechanisms for virtualizing the CPU in place. But a major question is left unanswered: which process should we run at a given time? It is this question that the scheduler must answer, and thus the next topic of our study.

References

[A79] "Alto User's Handbook"
Xerox Palo Alto Research Center, September 1979
Available: http://history-computer.com/Library/AltoUsersHandbook.pdf
An amazing system, way ahead of its time. Became famous because Steve Jobs visited, took notes, and built Lisa and eventually Mac.

[C+04] "Microreboot — A Technique for Cheap Recovery"
George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, Armando Fox
OSDI '04, San Francisco, CA, December 2004
An excellent paper pointing out how far one can go with reboot in building more robust systems.

[I11] "Intel 64 and IA-32 Architectures Software Developer's Manual"
Volume 3A and 3B: System Programming Guide
Intel Corporation, January 2011

[K+61] "One-Level Storage System"
T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner
IRE Transactions on Electronic Computers, April 1962
The Atlas pioneered much of what you see in modern systems. However, this paper is not the best one to read. If you were to only read one, you might try the historical perspective below [L78].

[L78] "The Manchester Mark I and Atlas: A Historical Perspective"
S. H. Lavington
Communications of the ACM, 21:1, January 1978
A history of the early development of computers and the pioneering efforts of Atlas.

[M+63] "A Time-Sharing Debugging System for a Small Computer"
J. McCarthy, S. Boilen, E. Fredkin, J. C. R. Licklider
AFIPS '63 (Spring), May, 1963, New York, USA
An early paper about time-sharing that refers to using a timer interrupt; the quote that discusses it: "The basic task of the channel 17 clock routine is to decide whether to remove the current user from core and if so to decide which user program to swap in as he goes out."

[MS96] "lmbench: Portable tools for performance analysis"
Larry McVoy and Carl Staelin
USENIX Annual Technical Conference, January 1996
A fun paper about how to measure a number of different things about your OS and its performance. Download lmbench and give it a try.

[M11] "Mac OS 9"
January 2011
Available: http://en.wikipedia.org/wiki/Mac_OS_9

[O90] "Why Aren't Operating Systems Getting Faster as Fast as Hardware?"
J. Ousterhout
USENIX Summer Conference, June 1990
A classic paper on the nature of operating system performance.

[P10] "The Single UNIX Specification, Version 3"
The Open Group, May 2010
Available: http://www.unix.org/version3/
This is hard and painful to read, so probably avoid it if you can.

[S07] "The Geometry of Innocent Flesh on the Bone:
Return-into-libc without Function Calls (on the x86)"
Hovav Shacham
CCS '07, October 2007
One of those awesome, mind-blowing ideas that you'll see in research from time to time. The author shows that if you can jump into code arbitrarily, you can essentially stitch together any code sequence you like (given a large code base); read the paper for the details. The technique makes it even harder to defend against malicious attacks, alas.

Homework (Measurement)

ASIDE: **MEASUREMENT HOMEWORKS**
Measurement homeworks are small exercises where you write code to run on a real machine, in order to measure some aspect of OS or hardware performance. The idea behind such homeworks is to give you a little bit of hands-on experience with a real operating system.

In this homework, you'll measure the costs of a system call and context switch. Measuring the cost of a system call is relatively easy. For example, you could repeatedly call a simple system call (e.g., performing a 0-byte read), and time how long it takes; dividing the time by the number of iterations gives you an estimate of the cost of a system call.

One thing you'll have to take into account is the precision and accuracy of your timer. A typical timer that you can use is `gettimeofday()`; read the man page for details. What you'll see there is that `gettimeofday()` returns the time in microseconds since 1970; however, this does not mean that the timer is precise to the microsecond. Measure back-to-back calls to `gettimeofday()` to learn something about how precise the timer really is; this will tell you how many iterations of your null system-call test you'll have to run in order to get a good measurement result. If `gettimeofday()` is not precise enough for you, you might look into using the `rdtsc` instruction available on x86 machines.

Measuring the cost of a context switch is a little trickier. The lmbench benchmark does so by running two processes on a single CPU, and setting up two UNIX pipes between them; a pipe is just one of many ways processes in a UNIX system can communicate with one another. The first process then issues a write to the first pipe, and waits for a read on the second; upon seeing the first process waiting for something to read from the second pipe, the OS puts the first process in the blocked state, and switches to the other process, which reads from the first pipe and then writes to the second. When the second process tries to read from the first pipe again, it blocks, and thus the back-and-forth cycle of communication continues. By measuring the cost of communicating like this repeatedly, lmbench can make a good estimate of the cost of a context switch. You can try to re-create something similar here, using pipes, or perhaps some other communication mechanism such as UNIX sockets.

One difficulty in measuring context-switch cost arises in systems with more than one CPU; what you need to do on such a system is ensure that your context-switching processes are located on the same processor. Fortunately, most operating systems have calls to bind a process to a particular processor; on Linux, for example, the `sched_setaffinity()` call is what you're looking for. By ensuring both processes are on the same processor, you are making sure to measure the cost of the OS stopping one process and restoring another on the same CPU.

7

Scheduling: Introduction

By now low-level **mechanisms** of running processes (e.g., context switching) should be clear; if they are not, go back a chapter or two, and read the description of how that stuff works again. However, we have yet to understand the high-level **policies** that an OS scheduler employs. We will now do just that, presenting a series of **scheduling policies** (sometimes called **disciplines**) that various smart and hard-working people have developed over the years.

The origins of scheduling, in fact, predate computer systems; early approaches were taken from the field of operations management and applied to computers. This reality should be no surprise: assembly lines and many other human endeavors also require scheduling, and many of the same concerns exist therein, including a laser-like desire for efficiency. And thus, our problem:

THE CRUX: HOW TO DEVELOP SCHEDULING POLICY

How should we develop a basic framework for thinking about scheduling policies? What are the key assumptions? What metrics are important? What basic approaches have been used in the earliest of computer systems?

7.1 Workload Assumptions

Before getting into the range of possible policies, let us first make a number of simplifying assumptions about the processes running in the system, sometimes collectively called the **workload**. Determining the workload is a critical part of building policies, and the more you know about workload, the more fine-tuned your policy can be.

The workload assumptions we make here are mostly unrealistic, but that is alright (for now), because we will relax them as we go, and eventually develop what we will refer to as ... *(dramatic pause) ...*

a **fully-operational scheduling discipline**[1].

We will make the following assumptions about the processes, sometimes called **jobs**, that are running in the system:

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known.

We said many of these assumptions were unrealistic, but just as some animals are more equal than others in Orwell's *Animal Farm* [O45], some assumptions are more unrealistic than others in this chapter. In particular, it might bother you that the run-time of each job is known: this would make the scheduler omniscient, which, although it would be great (probably), is not likely to happen anytime soon.

7.2 Scheduling Metrics

Beyond making workload assumptions, we also need one more thing to enable us to compare different scheduling policies: a **scheduling metric**. A metric is just something that we use to *measure* something, and there are a number of different metrics that make sense in scheduling.

For now, however, let us also simplify our life by simply having a single metric: **turnaround time**. The turnaround time of a job is defined as the time at which the job completes minus the time at which the job arrived in the system. More formally, the turnaround time $T_{turnaround}$ is:

$$T_{turnaround} = T_{completion} - T_{arrival} \tag{7.1}$$

Because we have assumed that all jobs arrive at the same time, for now $T_{arrival} = 0$ and hence $T_{turnaround} = T_{completion}$. This fact will change as we relax the aforementioned assumptions.

You should note that turnaround time is a **performance** metric, which will be our primary focus this chapter. Another metric of interest is **fairness**, as measured (for example) by **Jain's Fairness Index** [J91]. Performance and fairness are often at odds in scheduling; a scheduler, for example, may optimize performance but at the cost of preventing a few jobs from running, thus decreasing fairness. This conundrum shows us that life isn't always perfect.

7.3 First In, First Out (FIFO)

The most basic algorithm we can implement is known as **First In, First Out** (**FIFO**) scheduling or sometimes **First Come, First Served** (**FCFS**).

[1] Said in the same way you would say "A fully-operational Death Star."

FIFO has a number of positive properties: it is clearly simple and thus easy to implement. And, given our assumptions, it works pretty well.

Let's do a quick example together. Imagine three jobs arrive in the system, A, B, and C, at roughly the same time ($T_{arrival} = 0$). Because FIFO has to put some job first, let's assume that while they all arrived simultaneously, A arrived just a hair before B which arrived just a hair before C. Assume also that each job runs for 10 seconds. What will the **average turnaround time** be for these jobs?

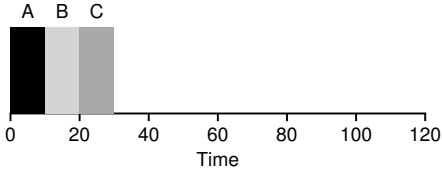


Figure 7.1: **FIFO Simple Example**

From Figure 7.1, you can see that A finished at 10, B at 20, and C at 30. Thus, the average turnaround time for the three jobs is simply $\frac{10+20+30}{3} = 20$. Computing turnaround time is as easy as that.

Now let's relax one of our assumptions. In particular, let's relax assumption 1, and thus no longer assume that each job runs for the same amount of time. How does FIFO perform now? What kind of workload could you construct to make FIFO perform poorly?

(think about this before reading on ... keep thinking ... got it?!)

Presumably you've figured this out by now, but just in case, let's do an example to show how jobs of different lengths can lead to trouble for FIFO scheduling. In particular, let's again assume three jobs (A, B, and C), but this time A runs for 100 seconds while B and C run for 10 each.

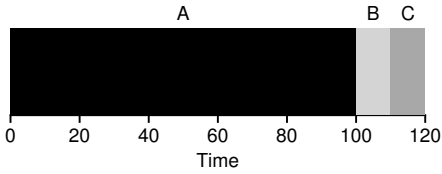


Figure 7.2: **Why FIFO Is Not That Great**

As you can see in Figure 7.2, Job A runs first for the full 100 seconds before B or C even get a chance to run. Thus, the average turnaround time for the system is high: a painful 110 seconds ($\frac{100+110+120}{3} = 110$).

This problem is generally referred to as the **convoy effect** [B+79], where a number of relatively-short potential consumers of a resource get queued behind a heavyweight resource consumer. This scheduling scenario might remind you of a single line at a grocery store and what you feel like when

TIP: THE PRINCIPLE OF SJF
Shortest Job First represents a general scheduling principle that can be applied to any system where the perceived turnaround time per customer (or, in our case, a job) matters. Think of any line you have waited in: if the establishment in question cares about customer satisfaction, it is likely they have taken SJF into account. For example, grocery stores commonly have a "ten-items-or-less" line to ensure that shoppers with only a few things to purchase don't get stuck behind the family preparing for some upcoming nuclear winter.

you see the person in front of you with three carts full of provisions and a checkbook out; it's going to be a while[2].

So what should we do? How can we develop a better algorithm to deal with our new reality of jobs that run for different amounts of time? Think about it first; then read on.

7.4 Shortest Job First (SJF)

It turns out that a very simple approach solves this problem; in fact it is an idea stolen from operations research [C54,PV56] and applied to scheduling of jobs in computer systems. This new scheduling discipline is known as **Shortest Job First (SJF)**, and the name should be easy to remember because it describes the policy quite completely: it runs the shortest job first, then the next shortest, and so on.

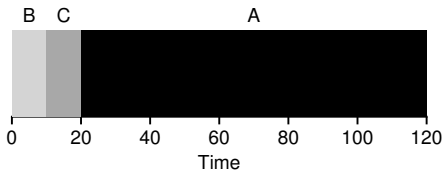


Figure 7.3: **SJF Simple Example**

Let's take our example above but with SJF as our scheduling policy. Figure 7.3 shows the results of running A, B, and C. Hopefully the diagram makes it clear why SJF performs much better with regards to average turnaround time. Simply by running B and C before A, SJF reduces average turnaround from 110 seconds to 50 ($\frac{10+20+120}{3} = 50$), more than a factor of two improvement.

In fact, given our assumptions about jobs all arriving at the same time, we could prove that SJF is indeed an **optimal** scheduling algorithm. How-

[2]Recommended action in this case: either quickly switch to a different line, or take a long, deep, and relaxing breath. That's right, breathe in, breathe out. It will be OK, don't worry.

ASIDE: **PREEMPTIVE SCHEDULERS**
In the old days of batch computing, a number of **non-preemptive** schedulers were developed; such systems would run each job to completion before considering whether to run a new job. Virtually all modern schedulers are **preemptive**, and quite willing to stop one process from running in order to run another. This implies that the scheduler employs the mechanisms we learned about previously; in particular, the scheduler can perform a **context switch**, stopping one running process temporarily and resuming (or starting) another.

ever, you are in a systems class, not theory or operations research; no proofs are allowed.

Thus we arrive upon a good approach to scheduling with SJF, but our assumptions are still fairly unrealistic. Let's relax another. In particular, we can target assumption 2, and now assume that jobs can arrive at any time instead of all at once. What problems does this lead to?

(Another pause to think ... are you thinking? Come on, you can do it)

Here we can illustrate the problem again with an example. This time, assume A arrives at $t = 0$ and needs to run for 100 seconds, whereas B and C arrive at $t = 10$ and each need to run for 10 seconds. With pure SJF, we'd get the schedule seen in Figure 7.4.

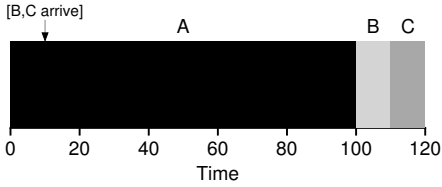


Figure 7.4: **SJF With Late Arrivals From B and C**

As you can see from the figure, even though B and C arrived shortly after A, they still are forced to wait until A has completed, and thus suffer the same convoy problem. Average turnaround time for these three jobs is 103.33 seconds ($\frac{100+(110-10)+(120-10)}{3}$). What can a scheduler do?

7.5 Shortest Time-to-Completion First (STCF)

To address this concern, we need to relax assumption 3 (that jobs must run to completion), so let's do that. We also need some machinery within the scheduler itself. As you might have guessed, given our previous discussion about timer interrupts and context switching, the scheduler can certainly do something else when B and C arrive: it can **preempt** job A and decide to run another job, perhaps continuing A later. SJF by our definition is a **non-preemptive** scheduler, and thus suffers from the problems described above.

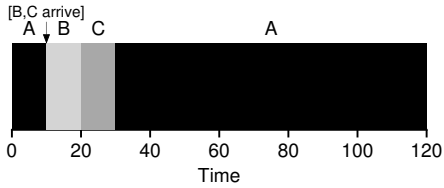


Figure 7.5: **STCF Simple Example**

Fortunately, there is a scheduler which does exactly that: add preemption to SJF, known as the **Shortest Time-to-Completion First** (**STCF**) or **Preemptive Shortest Job First** (**PSJF**) scheduler [CK68]. Any time a new job enters the system, the STCF scheduler determines which of the remaining jobs (including the new job) has the least time left, and schedules that one. Thus, in our example, STCF would preempt A and run B and C to completion; only when they are finished would A's remaining time be scheduled. Figure 7.5 shows an example.

The result is a much-improved average turnaround time: 50 seconds ($\frac{(120-0)+(20-10)+(30-10)}{3}$). And as before, given our new assumptions, STCF is provably optimal; given that SJF is optimal if all jobs arrive at the same time, you should probably be able to see the intuition behind the optimality of STCF.

7.6 A New Metric: Response Time

Thus, if we knew job lengths, and that jobs only used the CPU, and our only metric was turnaround time, STCF would be a great policy. In fact, for a number of early batch computing systems, these types of scheduling algorithms made some sense. However, the introduction of time-shared machines changed all that. Now users would sit at a terminal and demand interactive performance from the system as well. And thus, a new metric was born: **response time**.

Response time is defined as the time from when the job arrives in a system to the first time it is scheduled. More formally:

$$T_{response} = T_{firstrun} - T_{arrival} \tag{7.2}$$

For example, if we had the schedule above (with A arriving at time 0, and B and C at time 10), the response time of each job is as follows: 0 for job A, 0 for B, and 10 for C (average: 3.33).

As you might be thinking, STCF and related disciplines are not particularly good for response time. If three jobs arrive at the same time, for example, the third job has to wait for the previous two jobs to run *in their entirety* before being scheduled just once. While great for turnaround time, this approach is quite bad for response time and interactivity. Indeed, imagine sitting at a terminal, typing, and having to wait 10 seconds

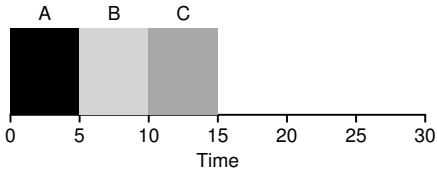


Figure 7.6: **SJF Again (Bad for Response Time)**

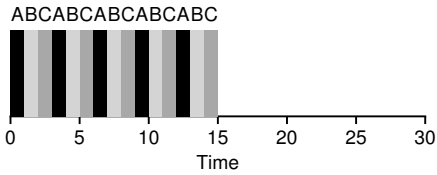


Figure 7.7: **Round Robin (Good for Response Time)**

to see a response from the system just because some other job got scheduled in front of yours: not too pleasant.

Thus, we are left with another problem: how can we build a scheduler that is sensitive to response time?

7.7 Round Robin

To solve this problem, we will introduce a new scheduling algorithm, classically referred to as **Round-Robin (RR)** scheduling [K64]. The basic idea is simple: instead of running jobs to completion, RR runs a job for a **time slice** (sometimes called a **scheduling quantum**) and then switches to the next job in the run queue. It repeatedly does so until the jobs are finished. For this reason, RR is sometimes called **time-slicing**. Note that the length of a time slice must be a multiple of the timer-interrupt period; thus if the timer interrupts every 10 milliseconds, the time slice could be 10, 20, or any other multiple of 10 ms.

To understand RR in more detail, let's look at an example. Assume three jobs A, B, and C arrive at the same time in the system, and that they each wish to run for 5 seconds. An SJF scheduler runs each job to completion before running another (Figure 7.6). In contrast, RR with a time-slice of 1 second would cycle through the jobs quickly (Figure 7.7).

The average response time of RR is: $\frac{0+1+2}{3} = 1$; for SJF, average response time is: $\frac{0+5+10}{3} = 5$.

As you can see, the length of the time slice is critical for RR. The shorter it is, the better the performance of RR under the response-time metric. However, making the time slice too short is problematic: suddenly the cost of context switching will dominate overall performance. Thus, deciding on the length of the time slice presents a trade-off to a system designer, making it long enough to **amortize** the cost of switching without making it so long that the system is no longer responsive.

TIP: AMORTIZATION CAN REDUCE COSTS
The general technique of **amortization** is commonly used in systems when there is a fixed cost to some operation. By incurring that cost less often (i.e., by performing the operation fewer times), the total cost to the system is reduced. For example, if the time slice is set to 10 ms, and the context-switch cost is 1 ms, roughly 10% of time is spent context switching and is thus wasted. If we want to *amortize* this cost, we can increase the time slice, e.g., to 100 ms. In this case, less than 1% of time is spent context switching, and thus the cost of time-slicing has been amortized.

Note that the cost of context switching does not arise solely from the OS actions of saving and restoring a few registers. When programs run, they build up a great deal of state in CPU caches, TLBs, branch predictors, and other on-chip hardware. Switching to another job causes this state to be flushed and new state relevant to the currently-running job to be brought in, which may exact a noticeable performance cost [MB91].

RR, with a reasonable time slice, is thus an excellent scheduler if response time is our only metric. But what about our old friend turnaround time? Let's look at our example above again. A, B, and C, each with running times of 5 seconds, arrive at the same time, and RR is the scheduler with a (long) 1-second time slice. We can see from the picture above that A finishes at 13, B at 14, and C at 15, for an average of 14. Pretty awful!

It is not surprising, then, that RR is indeed one of the *worst* policies if turnaround time is our metric. Intuitively, this should make sense: what RR is doing is stretching out each job as long as it can, by only running each job for a short bit before moving to the next. Because turnaround time only cares about when jobs finish, RR is nearly pessimal, even worse than simple FIFO in many cases.

More generally, any policy (such as RR) that is **fair**, i.e., that evenly divides the CPU among active processes on a small time scale, will perform poorly on metrics such as turnaround time. Indeed, this is an inherent trade-off: if you are willing to be unfair, you can run shorter jobs to completion, but at the cost of response time; if you instead value fairness, response time is lowered, but at the cost of turnaround time. This type of **trade-off** is common in systems; you can't have your cake and eat it too[3].

We have developed two types of schedulers. The first type (SJF, STCF) optimizes turnaround time, but is bad for response time. The second type (RR) optimizes response time but is bad for turnaround. And we still have two assumptions which need to be relaxed: assumption 4 (that jobs do no I/O), and assumption 5 (that the run-time of each job is known). Let's tackle those assumptions next.

[3] A saying that confuses people, because it should be "You can't *keep* your cake and eat it too" (which is kind of obvious, no?). Amazingly, there is a wikipedia page about this saying; even more amazingly, it is kind of fun to read [W15]. As they say in Italian, you can't *Avere la botte piena e la moglie ubriaca.*

TIP: OVERLAP ENABLES HIGHER UTILIZATION
When possible, **overlap** operations to maximize the utilization of systems. Overlap is useful in many different domains, including when performing disk I/O or sending messages to remote machines; in either case, starting the operation and then switching to other work is a good idea, and improves the overall utilization and efficiency of the system.

7.8 Incorporating I/O

First we will relax assumption 4 — of course all programs perform I/O. Imagine a program that didn't take any input: it would produce the same output each time. Imagine one without output: it is the proverbial tree falling in the forest, with no one to see it; it doesn't matter that it ran.

A scheduler clearly has a decision to make when a job initiates an I/O request, because the currently-running job won't be using the CPU during the I/O; it is **blocked** waiting for I/O completion. If the I/O is sent to a hard disk drive, the process might be blocked for a few milliseconds or longer, depending on the current I/O load of the drive. Thus, the scheduler should probably schedule another job on the CPU at that time.

The scheduler also has to make a decision when the I/O completes. When that occurs, an interrupt is raised, and the OS runs and moves the process that issued the I/O from blocked back to the ready state. Of course, it could even decide to run the job at that point. How should the OS treat each job?

To understand this issue better, let us assume we have two jobs, A and B, which each need 50 ms of CPU time. However, there is one obvious difference: A runs for 10 ms and then issues an I/O request (assume here that I/Os each take 10 ms), whereas B simply uses the CPU for 50 ms and performs no I/O. The scheduler runs A first, then B after (Figure 7.8).

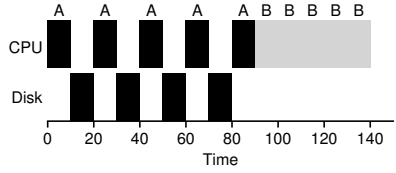


Figure 7.8: **Poor Use of Resources**

Assume we are trying to build a STCF scheduler. How should such a scheduler account for the fact that A is broken up into 5 10-ms sub-jobs, whereas B is just a single 50-ms CPU demand? Clearly, just running one job and then the other without considering how to take I/O into account makes little sense.

A common approach is to treat each 10-ms sub-job of A as an independent job. Thus, when the system starts, its choice is whether to schedule

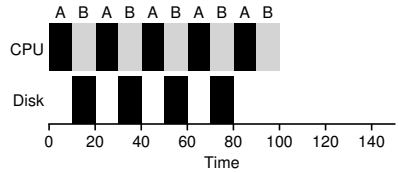


Figure 7.9: **Overlap Allows Better Use of Resources**

a 10-ms A or a 50-ms B. With STCF, the choice is clear: choose the shorter one, in this case A. Then, when the first sub-job of A has completed, only B is left, and it begins running. Then a new sub-job of A is submitted, and it preempts B and runs for 10 ms. Doing so allows for **overlap**, with the CPU being used by one process while waiting for the I/O of another process to complete; the system is thus better utilized (see Figure 7.9).

And thus we see how a scheduler might incorporate I/O. By treating each CPU burst as a job, the scheduler makes sure processes that are "interactive" get run frequently. While those interactive jobs are performing I/O, other CPU-intensive jobs run, thus better utilizing the processor.

7.9 No More Oracle

With a basic approach to I/O in place, we come to our final assumption: that the scheduler knows the length of each job. As we said before, this is likely the worst assumption we could make. In fact, in a general-purpose OS (like the ones we care about), the OS usually knows very little about the length of each job. Thus, how can we build an approach that behaves like SJF/STCF without such *a priori* knowledge? Further, how can we incorporate some of the ideas we have seen with the RR scheduler so that response time is also quite good?

7.10 Summary

We have introduced the basic ideas behind scheduling and developed two families of approaches. The first runs the shortest job remaining and thus optimizes turnaround time; the second alternates between all jobs and thus optimizes response time. Both are bad where the other is good, alas, an inherent trade-off common in systems. We have also seen how we might incorporate I/O into the picture, but have still not solved the problem of the fundamental inability of the OS to see into the future. Shortly, we will see how to overcome this problem, by building a scheduler that uses the recent past to predict the future. This scheduler is known as the **multi-level feedback queue**, and it is the topic of the next chapter.

References

[B+79] "The Convoy Phenomenon"
M. Blasgen, J. Gray, M. Mitoma, T. Price
ACM Operating Systems Review, 13:2, April 1979
Perhaps the first reference to convoys, which occurs in databases as well as the OS.

[C54] "Priority Assignment in Waiting Line Problems"
A. Cobham
Journal of Operations Research, 2:70, pages 70–76, 1954
The pioneering paper on using an SJF approach in scheduling the repair of machines.

[K64] "Analysis of a Time-Shared Processor"
Leonard Kleinrock
Naval Research Logistics Quarterly, 11:1, pages 59–73, March 1964
May be the first reference to the round-robin scheduling algorithm; certainly one of the first analyses of said approach to scheduling a time-shared system.

[CK68] "Computer Scheduling Methods and their Countermeasures"
Edward G. Coffman and Leonard Kleinrock
AFIPS '68 (Spring), April 1968
An excellent early introduction to and analysis of a number of basic scheduling disciplines.

[J91] "The Art of Computer Systems Performance Analysis:
Techniques for Experimental Design, Measurement, Simulation, and Modeling"
R. Jain
Interscience, New York, April 1991
The standard text on computer systems measurement. A great reference for your library, for sure.

[O45] "Animal Farm"
George Orwell
Secker and Warburg (London), 1945
A great but depressing allegorical book about power and its corruptions. Some say it is a critique of Stalin and the pre-WWII Stalin era in the U.S.S.R; we say it's a critique of pigs.

[PV56] "Machine Repair as a Priority Waiting-Line Problem"
Thomas E. Phipps Jr. and W. R. Van Voorhis
Operations Research, 4:1, pages 76–86, February 1956
Follow-on work that generalizes the SJF approach to machine repair from Cobham's original work; also postulates the utility of an STCF approach in such an environment. Specifically, "There are certain types of repair work, ... involving much dismantling and covering the floor with nuts and bolts, which certainly should not be interrupted once undertaken; in other cases it would be inadvisable to continue work on a long job if one or more short ones became available (p.81)."

[MB91] "The effect of context switches on cache performance"
Jeffrey C. Mogul and Anita Borg
ASPLOS, 1991
A nice study on how cache performance can be affected by context switching; less of an issue in today's systems where processors issue billions of instructions per second but context-switches still happen in the millisecond time range.

[W15] "You can't have your cake and eat it"
`http://en.wikipedia.org/wiki/You_can't_have_your_cake_and_eat_it`
Wikipedia, as of December 2015
The best part of this page is reading all the similar idioms from other languages. In Tamil, you can't "have both the moustache and drink the soup."

Homework

This program, `scheduler.py`, allows you to see how different schedulers perform under scheduling metrics such as response time, turnaround time, and total wait time. See the README for details.

Questions

1. Compute the response time and turnaround time when running three jobs of length 200 with the SJF and FIFO schedulers.
2. Now do the same but with jobs of different lengths: 100, 200, and 300.
3. Now do the same, but also with the RR scheduler and a time-slice of 1.
4. For what types of workloads does SJF deliver the same turnaround times as FIFO?
5. For what types of workloads and quantum lengths does SJF deliver the same response times as RR?
6. What happens to response time with SJF as job lengths increase? Can you use the simulator to demonstrate the trend?
7. What happens to response time with RR as quantum lengths increase? Can you write an equation that gives the worst-case response time, given N jobs?

8

Scheduling: The Multi-Level Feedback Queue

In this chapter, we'll tackle the problem of developing one of the most well-known approaches to scheduling, known as the **Multi-level Feedback Queue (MLFQ)**. The Multi-level Feedback Queue (MLFQ) scheduler was first described by Corbato et al. in 1962 [C+62] in a system known as the Compatible Time-Sharing System (CTSS), and this work, along with later work on Multics, led the ACM to award Corbato its highest honor, the **Turing Award**. The scheduler has subsequently been refined throughout the years to the implementations you will encounter in some modern systems.

The fundamental problem MLFQ tries to address is two-fold. First, it would like to optimize *turnaround time*, which, as we saw in the previous note, is done by running shorter jobs first; unfortunately, the OS doesn't generally know how long a job will run for, exactly the knowledge that algorithms like SJF (or STCF) require. Second, MLFQ would like to make a system feel responsive to interactive users (i.e., users sitting and staring at the screen, waiting for a process to finish), and thus minimize *response time*; unfortunately, algorithms like Round Robin reduce response time but are terrible for turnaround time. Thus, our problem: given that we in general do not know anything about a process, how can we build a scheduler to achieve these goals? How can the scheduler learn, as the system runs, the characteristics of the jobs it is running, and thus make better scheduling decisions?

THE CRUX:
HOW TO SCHEDULE WITHOUT PERFECT KNOWLEDGE?
How can we design a scheduler that both minimizes response time for interactive jobs while also minimizing turnaround time without *a priori* knowledge of job length?

TIP: LEARN FROM HISTORY
The multi-level feedback queue is an excellent example of a system that learns from the past to predict the future. Such approaches are common in operating systems (and many other places in Computer Science, including hardware branch predictors and caching algorithms). Such approaches work when jobs have phases of behavior and are thus predictable; of course, one must be careful with such techniques, as they can easily be wrong and drive a system to make worse decisions than they would have with no knowledge at all.

8.1 MLFQ: Basic Rules

To build such a scheduler, in this chapter we will describe the basic algorithms behind a multi-level feedback queue; although the specifics of many implemented MLFQs differ [E95], most approaches are similar.

In our treatment, the MLFQ has a number of distinct **queues**, each assigned a different **priority level**. At any given time, a job that is ready to run is on a single queue. MLFQ uses priorities to decide which job should run at a given time: a job with higher priority (i.e., a job on a higher queue) is chosen to run.

Of course, more than one job may be on a given queue, and thus have the *same* priority. In this case, we will just use round-robin scheduling among those jobs.

Thus, we arrive at the first two basic rules for MLFQ:

- **Rule 1:** If Priority(A) $>$ Priority(B), A runs (B doesn't).
- **Rule 2:** If Priority(A) $=$ Priority(B), A & B run in RR.

The key to MLFQ scheduling therefore lies in how the scheduler sets priorities. Rather than giving a fixed priority to each job, MLFQ *varies* the priority of a job based on its *observed behavior*. If, for example, a job repeatedly relinquishes the CPU while waiting for input from the keyboard, MLFQ will keep its priority high, as this is how an interactive process might behave. If, instead, a job uses the CPU intensively for long periods of time, MLFQ will reduce its priority. In this way, MLFQ will try to *learn* about processes as they run, and thus use the *history* of the job to predict its *future* behavior.

If we were to put forth a picture of what the queues might look like at a given instant, we might see something like the following (Figure 8.1). In the figure, two jobs (A and B) are at the highest priority level, while job C is in the middle and Job D is at the lowest priority. Given our current knowledge of how MLFQ works, the scheduler would just alternate time slices between A and B because they are the highest priority jobs in the system; poor jobs C and D would never even get to run — an outrage!

Of course, just showing a static snapshot of some queues does not really give you an idea of how MLFQ works. What we need is to under-

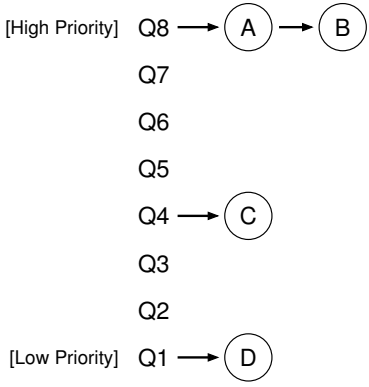


Figure 8.1: **MLFQ Example**

stand how job priority *changes* over time. And that, in a surprise only to those who are reading a chapter from this book for the first time, is exactly what we will do next.

8.2 Attempt #1: How To Change Priority

We now must decide how MLFQ is going to change the priority level of a job (and thus which queue it is on) over the lifetime of a job. To do this, we must keep in mind our workload: a mix of interactive jobs that are short-running (and may frequently relinquish the CPU), and some longer-running "CPU-bound" jobs that need a lot of CPU time but where response time isn't important. Here is our first attempt at a priority-adjustment algorithm:

- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4a:** If a job uses up an entire time slice while running, its priority is *reduced* (i.e., it moves down one queue).
- **Rule 4b:** If a job gives up the CPU before the time slice is up, it stays at the *same* priority level.

Example 1: A Single Long-Running Job

Let's look at some examples. First, we'll look at what happens when there has been a long running job in the system. Figure 8.2 shows what happens to this job over time in a three-queue scheduler.

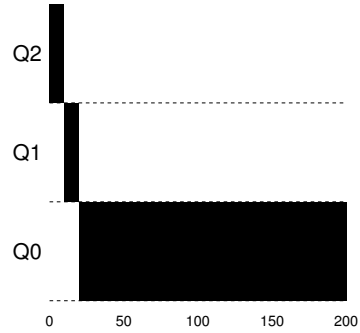


Figure 8.2: **Long-running Job Over Time**

As you can see in the example, the job enters at the highest priority (Q2). After a single time-slice of 10 ms, the scheduler reduces the job's priority by one, and thus the job is on Q1. After running at Q1 for a time slice, the job is finally lowered to the lowest priority in the system (Q0), where it remains. Pretty simple, no?

Example 2: Along Came A Short Job

Now let's look at a more complicated example, and hopefully see how MLFQ tries to approximate SJF. In this example, there are two jobs: A, which is a long-running CPU-intensive job, and B, which is a short-running interactive job. Assume A has been running for some time, and then B arrives. What will happen? Will MLFQ approximate SJF for B?

Figure 8.3 plots the results of this scenario. A (shown in black) is running along in the lowest-priority queue (as would any long-running CPU-intensive jobs); B (shown in gray) arrives at time $T = 100$, and thus is

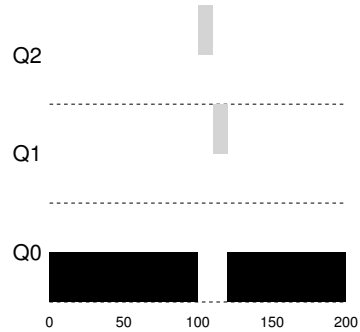


Figure 8.3: **Along Came An Interactive Job**

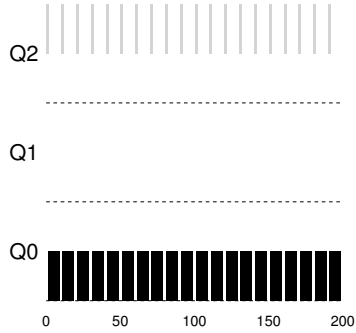


Figure 8.4: **A Mixed I/O-intensive and CPU-intensive Workload**

inserted into the highest queue; as its run-time is short (only 20 ms), B completes before reaching the bottom queue, in two time slices; then A resumes running (at low priority).

From this example, you can hopefully understand one of the major goals of the algorithm: because it doesn't *know* whether a job will be a short job or a long-running job, it first *assumes* it might be a short job, thus giving the job high priority. If it actually is a short job, it will run quickly and complete; if it is not a short job, it will slowly move down the queues, and thus soon prove itself to be a long-running more batch-like process. In this manner, MLFQ approximates SJF.

Example 3: What About I/O?

Let's now look at an example with some I/O. As Rule 4b states above, if a process gives up the processor before using up its time slice, we keep it at the same priority level. The intent of this rule is simple: if an interactive job, for example, is doing a lot of I/O (say by waiting for user input from the keyboard or mouse), it will relinquish the CPU before its time slice is complete; in such case, we don't wish to penalize the job and thus simply keep it at the same level.

Figure 8.4 shows an example of how this works, with an interactive job B (shown in gray) that needs the CPU only for 1 ms before performing an I/O competing for the CPU with a long-running batch job A (shown in black). The MLFQ approach keeps B at the highest priority because B keeps releasing the CPU; if B is an interactive job, MLFQ further achieves its goal of running interactive jobs quickly.

Problems With Our Current MLFQ

We thus have a basic MLFQ. It seems to do a fairly good job, sharing the CPU fairly between long-running jobs, and letting short or I/O-intensive interactive jobs run quickly. Unfortunately, the approach we have developed thus far contains serious flaws. Can you think of any?

(This is where you pause and think as deviously as you can)

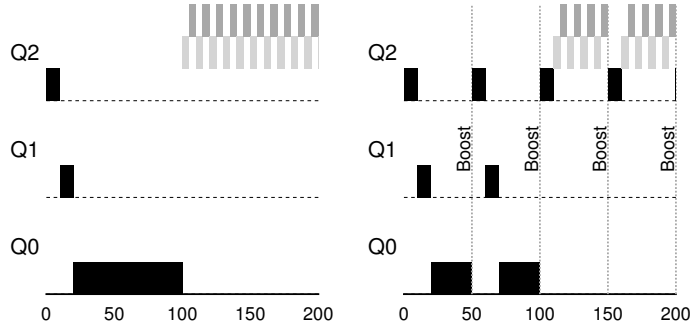


Figure 8.5: **Without (Left) and With (Right) Priority Boost**

First, there is the problem of **starvation**: if there are "too many" interactive jobs in the system, they will combine to consume *all* CPU time, and thus long-running jobs will *never* receive any CPU time (they **starve**). We'd like to make some progress on these jobs even in this scenario.

Second, a smart user could rewrite their program to **game the scheduler**. Gaming the scheduler generally refers to the idea of doing something sneaky to trick the scheduler into giving you more than your fair share of the resource. The algorithm we have described is susceptible to the following attack: before the time slice is over, issue an I/O operation (to some file you don't care about) and thus relinquish the CPU; doing so allows you to remain in the same queue, and thus gain a higher percentage of CPU time. When done right (e.g., by running for 99% of a time slice before relinquishing the CPU), a job could nearly monopolize the CPU.

Finally, a program may *change its behavior* over time; what was CPU-bound may transition to a phase of interactivity. With our current approach, such a job would be out of luck and not be treated like the other interactive jobs in the system.

8.3 Attempt #2: The Priority Boost

Let's try to change the rules and see if we can avoid the problem of starvation. What could we do in order to guarantee that CPU-bound jobs will make some progress (even if it is not much?).

The simple idea here is to periodically **boost** the priority of all the jobs in system. There are many ways to achieve this, but let's just do something simple: throw them all in the topmost queue; hence, a new rule:

- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

Our new rule solves two problems at once. First, processes are guaranteed not to starve: by sitting in the top queue, a job will share the CPU

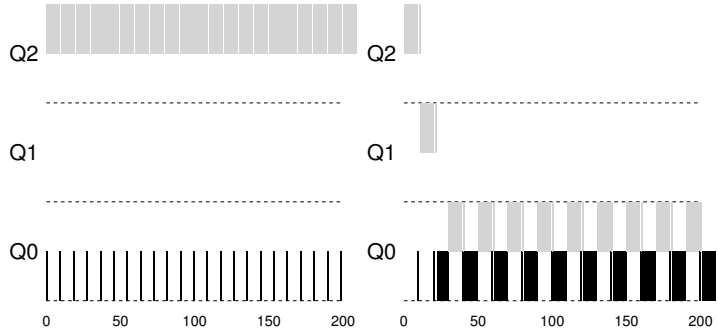


Figure 8.6: **Without (Left) and With (Right) Gaming Tolerance**

with other high-priority jobs in a round-robin fashion, and thus eventually receive service. Second, if a CPU-bound job has become interactive, the scheduler treats it properly once it has received the priority boost.

Let's see an example. In this scenario, we just show the behavior of a long-running job when competing for the CPU with two short-running interactive jobs. Two graphs are shown in Figure 8.5 (page 6). On the left, there is no priority boost, and thus the long-running job gets starved once the two short jobs arrive; on the right, there is a priority boost every 50 ms (which is likely too small of a value, but used here for the example), and thus we at least guarantee that the long-running job will make some progress, getting boosted to the highest priority every 50 ms and thus getting to run periodically.

Of course, the addition of the time period S leads to the obvious question: what should S be set to? John Ousterhout, a well-regarded systems researcher [O11], used to call such values in systems **voo-doo constants**, because they seemed to require some form of black magic to set them correctly. Unfortunately, S has that flavor. If it is set too high, long-running jobs could starve; too low, and interactive jobs may not get a proper share of the CPU.

8.4 Attempt #3: Better Accounting

We now have one more problem to solve: how to prevent gaming of our scheduler? The real culprit here, as you might have guessed, are Rules 4a and 4b, which let a job retain its priority by relinquishing the CPU before the time slice expires. So what should we do?

The solution here is to perform better **accounting** of CPU time at each level of the MLFQ. Instead of forgetting how much of a time slice a process used at a given level, the scheduler should keep track; once a process has used its allotment, it is demoted to the next priority queue. Whether

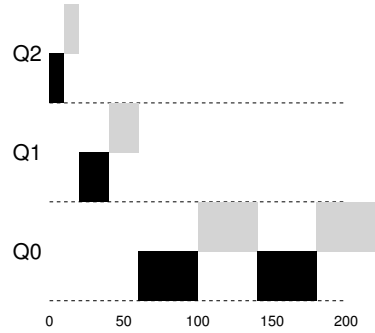


Figure 8.7: **Lower Priority, Longer Quanta**

it uses the time slice in one long burst or many small ones does not matter. We thus rewrite Rules 4a and 4b to the following single rule:

- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).

Let's look at an example. Figure 8.6 (page 7) shows what happens when a workload tries to game the scheduler with the old Rules 4a and 4b (on the left) as well the new anti-gaming Rule 4. Without any protection from gaming, a process can issue an I/O just before a time slice ends and thus dominate CPU time. With such protections in place, regardless of the I/O behavior of the process, it slowly moves down the queues, and thus cannot gain an unfair share of the CPU.

8.5 Tuning MLFQ And Other Issues

A few other issues arise with MLFQ scheduling. One big question is how to **parameterize** such a scheduler. For example, how many queues should there be? How big should the time slice be per queue? How often should priority be boosted in order to avoid starvation and account for changes in behavior? There are no easy answers to these questions, and thus only some experience with workloads and subsequent tuning of the scheduler will lead to a satisfactory balance.

For example, most MLFQ variants allow for varying time-slice length across different queues. The high-priority queues are usually given short time slices; they are comprised of interactive jobs, after all, and thus quickly alternating between them makes sense (e.g., 10 or fewer milliseconds). The low-priority queues, in contrast, contain long-running jobs that are CPU-bound; hence, longer time slices work well (e.g., 100s of ms). Figure 8.7 shows an example in which two long-running jobs run for 10 ms at the highest queue, 20 in the middle, and 40 at the lowest.

TIP: AVOID VOO-DOO CONSTANTS (OUSTERHOUT'S LAW)
Avoiding voo-doo constants is a good idea whenever possible. Unfortunately, as in the example above, it is often difficult. One could try to make the system learn a good value, but that too is not straightforward. The frequent result: a configuration file filled with default parameter values that a seasoned administrator can tweak when something isn't quite working correctly. As you can imagine, these are often left unmodified, and thus we are left to hope that the defaults work well in the field. This tip brought to you by our old OS professor, John Ousterhout, and hence we call it **Ousterhout's Law**.

The Solaris MLFQ implementation — the Time-Sharing scheduling class, or TS — is particularly easy to configure; it provides a set of tables that determine exactly how the priority of a process is altered throughout its lifetime, how long each time slice is, and how often to boost the priority of a job [AD00]; an administrator can muck with this table in order to make the scheduler behave in different ways. Default values for the table are 60 queues, with slowly increasing time-slice lengths from 20 milliseconds (highest priority) to a few hundred milliseconds (lowest), and priorities boosted around every 1 second or so.

Other MLFQ schedulers don't use a table or the exact rules described in this chapter; rather they adjust priorities using mathematical formulae. For example, the FreeBSD scheduler (version 4.3) uses a formula to calculate the current priority level of a job, basing it on how much CPU the process has used [LM+89]; in addition, usage is decayed over time, providing the desired priority boost in a different manner than described herein. See Epema's paper for an excellent overview of such **decay-usage** algorithms and their properties [E95].

Finally, many schedulers have a few other features that you might encounter. For example, some schedulers reserve the highest priority levels for operating system work; thus typical user jobs can never obtain the highest levels of priority in the system. Some systems also allow some user **advice** to help set priorities; for example, by using the command-line utility `nice` you can increase or decrease the priority of a job (somewhat) and thus increase or decrease its chances of running at any given time. See the man page for more.

8.6 MLFQ: Summary

We have described a scheduling approach known as the Multi-Level Feedback Queue (MLFQ). Hopefully you can now see why it is called that: it has *multiple levels* of queues, and uses *feedback* to determine the priority of a given job. History is its guide: pay attention to how jobs behave over time and treat them accordingly.

TIP: USE ADVICE WHERE POSSIBLE

As the operating system rarely knows what is best for each and every process of the system, it is often useful to provide interfaces to allow users or administrators to provide some **hints** to the OS. We often call such hints **advice**, as the OS need not necessarily pay attention to it, but rather might take the advice into account in order to make a better decision. Such hints are useful in many parts of the OS, including the scheduler (e.g., with `nice`), memory manager (e.g., `madvise`), and file system (e.g., informed prefetching and caching [P+95]).

The refined set of MLFQ rules, spread throughout the chapter, are reproduced here for your viewing pleasure:

- **Rule 1:** If Priority(A) $>$ Priority(B), A runs (B doesn't).
- **Rule 2:** If Priority(A) $=$ Priority(B), A & B run in RR.
- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

MLFQ is interesting for the following reason: instead of demanding *a priori* knowledge of the nature of a job, it observes the execution of a job and prioritizes it accordingly. In this way, it manages to achieve the best of both worlds: it can deliver excellent overall performance (similar to SJF/STCF) for short-running interactive jobs, and is fair and makes progress for long-running CPU-intensive workloads. For this reason, many systems, including BSD UNIX derivatives [LM+89, B86], Solaris [M06], and Windows NT and subsequent Windows operating systems [CS97] use a form of MLFQ as their base scheduler.

References

[AD00] "Multilevel Feedback Queue Scheduling in Solaris"
Andrea Arpaci-Dusseau
Available: http://www.cs.wisc.edu/~remzi/solaris-notes.pdf
A great short set of notes by one of the authors on the details of the Solaris scheduler. OK, we are probably biased in this description, but the notes are pretty darn good.

[B86] "The Design of the UNIX Operating System"
M.J. Bach
Prentice-Hall, 1986
One of the classic old books on how a real UNIX operating system is built; a definite must-read for kernel hackers.

[C+62] "An Experimental Time-Sharing System"
F. J. Corbato, M. M. Daggett, R. C. Daley
IFIPS 1962
A bit hard to read, but the source of many of the first ideas in multi-level feedback scheduling. Much of this later went into Multics, which one could argue was the most influential operating system of all time.

[CS97] "Inside Windows NT"
Helen Custer and David A. Solomon
Microsoft Press, 1997
The NT book, if you want to learn about something other than UNIX. Of course, why would you? OK, we're kidding; you might actually work for Microsoft some day you know.

[E95] "An Analysis of Decay-Usage Scheduling in Multiprocessors"
D.H.J. Epema
SIGMETRICS '95
A nice paper on the state of the art of scheduling back in the mid 1990s, including a good overview of the basic approach behind decay-usage schedulers.

[LM+89] "The Design and Implementation of the 4.3BSD UNIX Operating System"
S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman
Addison-Wesley, 1989
Another OS classic, written by four of the main people behind BSD. The later versions of this book, while more up to date, don't quite match the beauty of this one.

[M06] "Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture"
Richard McDougall
Prentice-Hall, 2006
A good book about Solaris and how it works.

[O11] "John Ousterhout's Home Page"
John Ousterhout
Available: http://www.stanford.edu/~ouster/
The home page of the famous Professor Ousterhout. The two co-authors of this book had the pleasure of taking graduate operating systems from Ousterhout while in graduate school; indeed, this is where the two co-authors got to know each other, eventually leading to marriage, kids, and even this book. Thus, you really can blame Ousterhout for this entire mess you're in.

[P+95] "Informed Prefetching and Caching"
R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, J. Zelenka
SOSP '95
A fun paper about some very cool ideas in file systems, including how applications can give the OS advice about what files it is accessing and how it plans to access them.

Homework

This program, `mlfq.py`, allows you to see how the MLFQ scheduler presented in this chapter behaves. See the README for details.

Questions

1. Run a few randomly-generated problems with just two jobs and two queues; compute the MLFQ execution trace for each. Make your life easier by limiting the length of each job and turning off I/Os.
2. How would you run the scheduler to reproduce each of the examples in the chapter?
3. How would you configure the scheduler parameters to behave just like a round-robin scheduler?
4. Craft a workload with two jobs and scheduler parameters so that one job takes advantage of the older Rules 4a and 4b (turned on with the `-S` flag) to game the scheduler and obtain 99% of the CPU over a particular time interval.
5. Given a system with a quantum length of 10 ms in its highest queue, how often would you have to boost jobs back to the highest priority level (with the `-B` flag) in order to guarantee that a single long-running (and potentially-starving) job gets at least 5% of the CPU?
6. One question that arises in scheduling is which end of a queue to add a job that just finished I/O; the `-I` flag changes this behavior for this scheduling simulator. Play around with some workloads and see if you can see the effect of this flag.

9

Scheduling: Proportional Share

In this chapter, we'll examine a different type of scheduler known as a **proportional-share** scheduler, also sometimes referred to as a **fair-share** scheduler. Proportional-share is based around a simple concept: instead of optimizing for turnaround or response time, a scheduler might instead try to guarantee that each job obtain a certain percentage of CPU time.

An excellent modern example of proportional-share scheduling is found in research by Waldspurger and Weihl [WW94], and is known as **lottery scheduling**; however, the idea is certainly much older [KL88]. The basic idea is quite simple: every so often, hold a lottery to determine which process should get to run next; processes that should run more often should be given more chances to win the lottery. Easy, no? Now, onto the details! But not before our crux:

CRUX: HOW TO SHARE THE CPU PROPORTIONALLY

How can we design a scheduler to share the CPU in a proportional manner? What are the key mechanisms for doing so? How effective are they?

9.1 Basic Concept: Tickets Represent Your Share

Underlying lottery scheduling is one very basic concept: **tickets**, which are used to represent the share of a resource that a process (or user or whatever) should receive. The percent of tickets that a process has represents its share of the system resource in question.

Let's look at an example. Imagine two processes, A and B, and further that A has 75 tickets while B has only 25. Thus, what we would like is for A to receive 75% of the CPU and B the remaining 25%.

Lottery scheduling achieves this probabilistically (but not deterministically) by holding a lottery every so often (say, every time slice). Holding a lottery is straightforward: the scheduler must know how many total tickets there are (in our example, there are 100). The scheduler then picks

TIP: USE RANDOMNESS

One of the most beautiful aspects of lottery scheduling is its use of **randomness**. When you have to make a decision, using such a randomized approach is often a robust and simple way of doing so.

Random approaches has at least three advantages over more traditional decisions. First, random often avoids strange corner-case behaviors that a more traditional algorithm may have trouble handling. For example, consider the LRU replacement policy (studied in more detail in a future chapter on virtual memory); while often a good replacement algorithm, LRU performs pessimally for some cyclic-sequential workloads. Random, on the other hand, has no such worst case.

Second, random also is lightweight, requiring little state to track alternatives. In a traditional fair-share scheduling algorithm, tracking how much CPU each process has received requires per-process accounting, which must be updated after running each process. Doing so randomly necessitates only the most minimal of per-process state (e.g., the number of tickets each has).

Finally, random can be quite fast. As long as generating a random number is quick, making the decision is also, and thus random can be used in a number of places where speed is required. Of course, the faster the need, the more random tends towards pseudo-random.

a winning ticket, which is a number from 0 to 99[1]. Assuming A holds tickets 0 through 74 and B 75 through 99, the winning ticket simply determines whether A or B runs. The scheduler then loads the state of that winning process and runs it.

Here is an example output of a lottery scheduler's winning tickets:

```
63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43  0 49 49
```

Here is the resulting schedule:

```
A     A  A     A  A  A  A  A  A     A     A  A  A  A  A  A
   B        B                    B     B
```

As you can see from the example, the use of randomness in lottery scheduling leads to a probabilistic correctness in meeting the desired proportion, but no guarantee. In our example above, B only gets to run 4 out of 20 time slices (20%), instead of the desired 25% allocation. However, the longer these two jobs compete, the more likely they are to achieve the desired percentages.

[1] Computer Scientists always start counting at 0. It is so odd to non-computer-types that famous people have felt obliged to write about why we do it this way [D82].

TIP: USE TICKETS TO REPRESENT SHARES

One of the most powerful (and basic) mechanisms in the design of lottery (and stride) scheduling is that of the **ticket**. The ticket is used to represent a process's share of the CPU in these examples, but can be applied much more broadly. For example, in more recent work on virtual memory management for hypervisors, Waldspurger shows how tickets can be used to represent a guest operating system's share of memory [W02]. Thus, if you are ever in need of a mechanism to represent a proportion of ownership, this concept just might be ... (wait for it) ... the ticket.

9.2 Ticket Mechanisms

Lottery scheduling also provides a number of mechanisms to manipulate tickets in different and sometimes useful ways. One way is with the concept of **ticket currency**. Currency allows a user with a set of tickets to allocate tickets among their own jobs in whatever currency they would like; the system then automatically converts said currency into the correct global value.

For example, assume users A and B have each been given 100 tickets. User A is running two jobs, A1 and A2, and gives them each 500 tickets (out of 1000 total) in User A's own currency. User B is running only 1 job and gives it 10 tickets (out of 10 total). The system will convert A1's and A2's allocation from 500 each in A's currency to 50 each in the global currency; similarly, B1's 10 tickets will be converted to 100 tickets. The lottery will then be held over the global ticket currency (200 total) to determine which job runs.

```
User A -> 500 (A's currency) to A1 ->  50 (global currency)
       -> 500 (A's currency) to A2 ->  50 (global currency)
User B ->  10 (B's currency) to B1 -> 100 (global currency)
```

Another useful mechanism is **ticket transfer**. With transfers, a process can temporarily hand off its tickets to another process. This ability is especially useful in a client/server setting, where a client process sends a message to a server asking it to do some work on the client's behalf. To speed up the work, the client can pass the tickets to the server and thus try to maximize the performance of the server while the server is handling the client's request. When finished, the server then transfers the tickets back to the client and all is as before.

Finally, **ticket inflation** can sometimes be a useful technique. With inflation, a process can temporarily raise or lower the number of tickets it owns. Of course, in a competitive scenario with processes that do not trust one another, this makes little sense; one greedy process could give itself a vast number of tickets and take over the machine. Rather, inflation can be applied in an environment where a group of processes trust one another; in such a case, if any one process knows it needs more CPU time, it can boost its ticket value as a way to reflect that need to the system, all without communicating with any other processes.

```
1  // counter: used to track if we've found the winner yet
2  int counter = 0;
3
4  // winner: use some call to a random number generator to
5  //         get a value, between 0 and the total # of tickets
6  int winner = getrandom(0, totaltickets);
7
8  // current: use this to walk through the list of jobs
9  node_t *current = head;
10
11 // loop until the sum of ticket values is > the winner
12 while (current) {
13     counter = counter + current->tickets;
14     if (counter > winner)
15         break; // found the winner
16     current = current->next;
17 }
18 // 'current' is the winner: schedule it...
```

Figure 9.1: **Lottery Scheduling Decision Code**

9.3 Implementation

Probably the most amazing thing about lottery scheduling is the simplicity of its implementation. All you need is a good random number generator to pick the winning ticket, a data structure to track the processes of the system (e.g., a list), and the total number of tickets.

Let's assume we keep the processes in a list. Here is an example comprised of three processes, A, B, and C, each with some number of tickets.

To make a scheduling decision, we first have to pick a random number (the winner) from the total number of tickets (400)[2] Let's say we pick the number 300. Then, we simply traverse the list, with a simple counter used to help us find the winner (Figure 9.1).

The code walks the list of processes, adding each ticket value to `counter` until the value exceeds `winner`. Once that is the case, the current list element is the winner. With our example of the winning ticket being 300, the following takes place. First, `counter` is incremented to 100 to account for A's tickets; because 100 is less than 300, the loop continues. Then `counter` would be updated to 150 (B's tickets), still less than 300 and thus again we continue. Finally, `counter` is updated to 400 (clearly greater than 300), and thus we break out of the loop with `current` pointing at C (the winner).

To make this process most efficient, it might generally be best to organize the list in sorted order, from the highest number of tickets to the

[2]Surprisingly, as pointed out by Björn Lindberg, this can be challenging to do correctly; for more details, see `http://stackoverflow.com/questions/2509679/how-to-generate-a-random-number-from-within-a-range`.

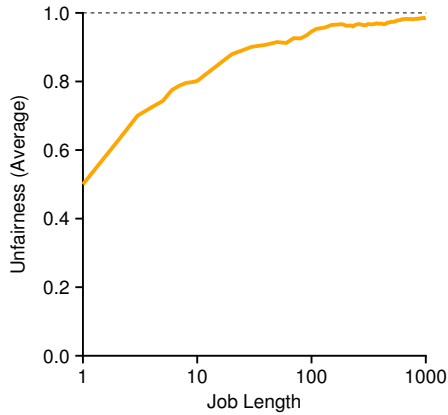


Figure 9.2: **Lottery Fairness Study**

lowest. The ordering does not affect the correctness of the algorithm; however, it does ensure in general that the fewest number of list iterations are taken, especially if there are a few processes that possess most of the tickets.

9.4 An Example

To make the dynamics of lottery scheduling more understandable, we now perform a brief study of the completion time of two jobs competing against one another, each with the same number of tickets (100) and same run time (R, which we will vary).

In this scenario, we'd like for each job to finish at roughly the same time, but due to the randomness of lottery scheduling, sometimes one job finishes before the other. To quantify this difference, we define a simple **unfairness metric**, U which is simply the time the first job completes divided by the time that the second job completes. For example, if $R = 10$, and the first job finishes at time 10 (and the second job at 20), $U = \frac{10}{20} = 0.5$. When both jobs finish at nearly the same time, U will be quite close to 1. In this scenario, that is our goal: a perfectly fair scheduler would achieve $U = 1$.

Figure 9.2 plots the average unfairness as the length of the two jobs (R) is varied from 1 to 1000 over thirty trials (results are generated via the simulator provided at the end of the chapter). As you can see from the graph, when the job length is not very long, average unfairness can be quite severe. Only as the jobs run for a significant number of time slices does the lottery scheduler approach the desired outcome.

9.5 How To Assign Tickets?

One problem we have not addressed with lottery scheduling is: how to assign tickets to jobs? This problem is a tough one, because of course how the system behaves is strongly dependent on how tickets are allocated. One approach is to assume that the users know best; in such a case, each user is handed some number of tickets, and a user can allocate tickets to any jobs they run as desired. However, this solution is a non-solution: it really doesn't tell you what to do. Thus, given a set of jobs, the "ticket-assignment problem" remains open.

9.6 Why Not Deterministic?

You might also be wondering: why use randomness at all? As we saw above, while randomness gets us a simple (and approximately correct) scheduler, it occasionally will not deliver the exact right proportions, especially over short time scales. For this reason, Waldspurger invented **stride scheduling**, a deterministic fair-share scheduler [W95].

Stride scheduling is also straightforward. Each job in the system has a stride, which is inverse in proportion to the number of tickets it has. In our example above, with jobs A, B, and C, with 100, 50, and 250 tickets, respectively, we can compute the stride of each by dividing some large number by the number of tickets each process has been assigned. For example, if we divide 10,000 by each of those ticket values, we obtain the following stride values for A, B, and C: 100, 200, and 40. We call this value the **stride** of each process; every time a process runs, we will increment a counter for it (called its **pass** value) by its stride to track its global progress.

The scheduler then uses the stride and pass to determine which process should run next. The basic idea is simple: at any given time, pick the process to run that has the lowest pass value so far; when you run a process, increment its pass counter by its stride. A pseudocode implementation is provided by Waldspurger [W95]:

```
current = remove_min(queue);       // pick client with minimum pass
schedule(current);                 // use resource for quantum
current->pass += current->stride; // compute next pass using stride
insert(queue, current);            // put back into the queue
```

In our example, we start with three processes (A, B, and C), with stride values of 100, 200, and 40, and all with pass values initially at 0. Thus, at first, any of the processes might run, as their pass values are equally low. Assume we pick A (arbitrarily; any of the processes with equal low pass values can be chosen). A runs; when finished with the time slice, we update its pass value to 100. Then we run B, whose pass value is then set to 200. Finally, we run C, whose pass value is incremented to 40. At this point, the algorithm will pick the lowest pass value, which is C's, and run it, updating its pass to 80 (C's stride is 40, as you recall). Then C will

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

Figure 9.3: **Stride Scheduling: A Trace**

run again (still the lowest pass value), raising its pass to 120. A will run now, updating its pass to 200 (now equal to B's). Then C will run twice more, updating its pass to 160 then 200. At this point, all pass values are equal again, and the process will repeat, ad infinitum. Figure 9.3 traces the behavior of the scheduler over time.

As we can see from the figure, C ran five times, A twice, and B just once, exactly in proportion to their ticket values of 250, 100, and 50. Lottery scheduling achieves the proportions probabilistically over time; stride scheduling gets them exactly right at the end of each scheduling cycle.

So you might be wondering: given the precision of stride scheduling, why use lottery scheduling at all? Well, lottery scheduling has one nice property that stride scheduling does not: no global state. Imagine a new job enters in the middle of our stride scheduling example above; what should its pass value be? Should it be set to 0? If so, it will monopolize the CPU. With lottery scheduling, there is no global state per process; we simply add a new process with whatever tickets it has, update the single global variable to track how many total tickets we have, and go from there. In this way, lottery makes it much easier to incorporate new processes in a sensible manner.

9.7 Summary

We have introduced the concept of proportional-share scheduling and briefly discussed two implementations: lottery and stride scheduling. Lottery uses randomness in a clever way to achieve proportional share; stride does so deterministically. Although both are conceptually interesting, they have not achieved wide-spread adoption as CPU schedulers for a variety of reasons. One is that such approaches do not particularly mesh well with I/O [AC97]; another is that they leave open the hard problem of ticket assignment, i.e., how do you know how many tickets your browser should be allocated? General-purpose schedulers (such as the MLFQ we discussed previously, and other similar Linux schedulers) do so more gracefully and thus are more widely deployed.

As a result, proportional-share schedulers are more useful in domains where some of these problems (such as assignment of shares) are relatively easy to solve. For example, in a **virtualized** data center, where you might like to assign one-quarter of your CPU cycles to the Windows VM and the rest to your base Linux installation, proportional sharing can be simple and effective. See Waldspurger [W02] for further details on how such a scheme is used to proportionally share memory in VMWare's ESX Server.

References

[AC97] "Extending Proportional-Share Scheduling to a Network of Workstations"
Andrea C. Arpaci-Dusseau and David E. Culler
PDPTA'97, June 1997
A paper by one of the authors on how to extend proportional-share scheduling to work better in a clustered environment.

[D82] "Why Numbering Should Start At Zero"
Edsger Dijkstra, August 1982
http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF
A short note from E. Dijkstra, one of the pioneers of computer science. We'll be hearing much more on this guy in the section on Concurrency. In the meanwhile, enjoy this note, which includes this motivating quote: "One of my colleagues — not a computing scientist — accused a number of younger computing scientists of 'pedantry' because they started numbering at zero." The note explains why doing so is logical.

[KL88] "A Fair Share Scheduler"
J. Kay and P. Lauder
CACM, Volume 31 Issue 1, January 1988
An early reference to a fair-share scheduler.

[WW94] "Lottery Scheduling: Flexible Proportional-Share Resource Management"
Carl A. Waldspurger and William E. Weihl
OSDI '94, November 1994
The landmark paper on lottery scheduling that got the systems community re-energized about scheduling, fair sharing, and the power of simple randomized algorithms.

[W95] "Lottery and Stride Scheduling: Flexible
Proportional-Share Resource Management"
Carl A. Waldspurger
Ph.D. Thesis, MIT, 1995
The award-winning thesis of Waldspurger's that outlines lottery and stride scheduling. If you're thinking of writing a Ph.D. dissertation at some point, you should always have a good example around, to give you something to strive for: this is such a good one.

[W02] "Memory Resource Management in VMware ESX Server"
Carl A. Waldspurger
OSDI '02, Boston, Massachusetts
The paper to read about memory management in VMMs (a.k.a., hypervisors). In addition to being relatively easy to read, the paper contains numerous cool ideas about this new type of VMM-level memory management.

Homework

This program, `lottery.py`, allows you to see how a lottery scheduler works. See the README for details.

Questions

1. Compute the solutions for simulations with 3 jobs and random seeds of 1, 2, and 3.
2. Now run with two specific jobs: each of length 10, but one (job 0) with just 1 ticket and the other (job 1) with 100 (e.g., `-l 10:1,10:100`). What happens when the number of tickets is so imbalanced? Will job 0 ever run before job 1 completes? How often? In general, what does such a ticket imbalance do to the behavior of lottery scheduling?
3. When running with two jobs of length 100 and equal ticket allocations of 100 (`-l 100:100,100:100`), how unfair is the scheduler? Run with some different random seeds to determine the (probabilistic) answer; let unfairness be determined by how much earlier one job finishes than the other.
4. How does your answer to the previous question change as the quantum size (`-q`) gets larger?
5. Can you make a version of the graph that is found in the chapter? What else would be worth exploring? How would the graph look with a stride scheduler?

10

Multiprocessor Scheduling (Advanced)

This chapter will introduce the basics of **multiprocessor scheduling**. As this topic is relatively advanced, it may be best to cover it *after* you have studied the topic of concurrency in some detail (i.e., the second major "easy piece" of the book).

After years of existence only in the high-end of the computing spectrum, **multiprocessor** systems are increasingly commonplace, and have found their way into desktop machines, laptops, and even mobile devices. The rise of the **multicore** processor, in which multiple CPU cores are packed onto a single chip, is the source of this proliferation; these chips have become popular as computer architects have had a difficult time making a single CPU much faster without using (way) too much power. And thus we all now have a few CPUs available to us, which is a good thing, right?

Of course, there are many difficulties that arise with the arrival of more than a single CPU. A primary one is that a typical application (i.e., some C program you wrote) only uses a single CPU; adding more CPUs does not make that single application run faster. To remedy this problem, you'll have to rewrite your application to run in **parallel**, perhaps using **threads** (as discussed in great detail in the second piece of this book). Multithreaded applications can spread work across multiple CPUs and thus run faster when given more CPU resources.

ASIDE: **ADVANCED CHAPTERS**

Advanced chapters require material from a broad swath of the book to truly understand, while logically fitting into a section that is earlier than said set of prerequisite materials. For example, this chapter on multiprocessor scheduling makes much more sense if you've first read the middle piece on concurrency; however, it logically fits into the part of the book on virtualization (generally) and CPU scheduling (specifically). Thus, it is recommended such chapters be covered out of order; in this case, after the second piece of the book.

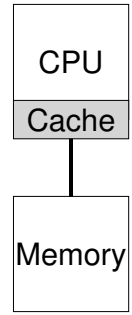


Figure 10.1: **Single CPU With Cache**

Beyond applications, a new problem that arises for the operating system is (not surprisingly!) that of **multiprocessor scheduling**. Thus far we've discussed a number of principles behind single-processor scheduling; how can we extend those ideas to work on multiple CPUs? What new problems must we overcome? And thus, our problem:

CRUX: HOW TO SCHEDULE JOBS ON MULTIPLE CPUS
How should the OS schedule jobs on multiple CPUs? What new problems arise? Do the same old techniques work, or are new ideas required?

10.1 Background: Multiprocessor Architecture

To understand the new issues surrounding multiprocessor scheduling, we have to understand a new and fundamental difference between single-CPU hardware and multi-CPU hardware. This difference centers around the use of hardware **caches** (e.g., Figure 10.1), and exactly how data is shared across multiple processors. We now discuss this issue further, at a high level. Details are available elsewhere [CSG99], in particular in an upper-level or perhaps graduate computer architecture course.

In a system with a single CPU, there are a hierarchy of **hardware caches** that in general help the processor run programs faster. Caches are small, fast memories that (in general) hold copies of *popular* data that is found in the main memory of the system. Main memory, in contrast, holds *all* of the data, but access to this larger memory is slower. By keeping frequently accessed data in a cache, the system can make the large, slow memory appear to be a fast one.

As an example, consider a program that issues an explicit load instruction to fetch a value from memory, and a simple system with only a single CPU; the CPU has a small cache (say 64 KB) and a large main memory.

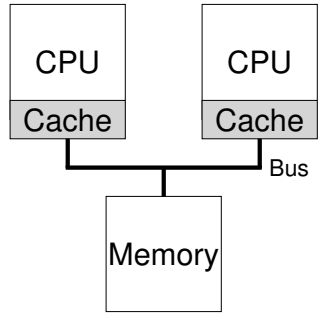


Figure 10.2: **Two CPUs With Caches Sharing Memory**

The first time a program issues this load, the data resides in main memory, and thus takes a long time to fetch (perhaps in the tens of nanoseconds, or even hundreds). The processor, anticipating that the data may be reused, puts a copy of the loaded data into the CPU cache. If the program later fetches this same data item again, the CPU first checks for it in the cache; if it finds it there, the data is fetched much more quickly (say, just a few nanoseconds), and thus the program runs faster.

Caches are thus based on the notion of **locality**, of which there are two kinds: **temporal locality** and **spatial locality**. The idea behind temporal locality is that when a piece of data is accessed, it is likely to be accessed again in the near future; imagine variables or even instructions themselves being accessed over and over again in a loop. The idea behind spatial locality is that if a program accesses a data item at address x, it is likely to access data items near x as well; here, think of a program streaming through an array, or instructions being executed one after the other. Because locality of these types exist in many programs, hardware systems can make good guesses about which data to put in a cache and thus work well.

Now for the tricky part: what happens when you have multiple processors in a single system, with a single shared main memory, as we see in Figure 10.2?

As it turns out, caching with multiple CPUs is much more complicated. Imagine, for example, that a program running on CPU 1 reads a data item (with value D) at address A; because the data is not in the cache on CPU 1, the system fetches it from main memory, and gets the value D. The program then modifies the value at address A, just updating its cache with the new value D'; writing the data through all the way to main memory is slow, so the system will (usually) do that later. Then assume the OS decides to stop running the program and move it to CPU 2. The program then re-reads the value at address A; there is no such data

CPU 2's cache, and thus the system fetches the value from main memory, and gets the old value D instead of the correct value D'. Oops!

This general problem is called the problem of **cache coherence**, and there is a vast research literature that describes many different subtleties involved with solving the problem [SHW11]. Here, we will skip all of the nuance and make some major points; take a computer architecture class (or three) to learn more.

The basic solution is provided by the hardware: by monitoring memory accesses, hardware can ensure that basically the "right thing" happens and that the view of a single shared memory is preserved. One way to do this on a bus-based system (as described above) is to use an old technique known as **bus snooping** [G83]; each cache pays attention to memory updates by observing the bus that connects them to main memory. When a CPU then sees an update for a data item it holds in its cache, it will notice the change and either **invalidate** its copy (i.e., remove it from its own cache) or **update** it (i.e., put the new value into its cache too). Write-back caches, as hinted at above, make this more complicated (because the write to main memory isn't visible until later), but you can imagine how the basic scheme might work.

10.2 Don't Forget Synchronization

Given that the caches do all of this work to provide coherence, do programs (or the OS itself) have to worry about anything when they access shared data? The answer, unfortunately, is yes, and is documented in great detail in the second piece of this book on the topic of concurrency. While we won't get into the details here, we'll sketch/review some of the basic ideas here (assuming you're familiar with concurrency).

When accessing (and in particular, updating) shared data items or structures across CPUs, mutual exclusion primitives (such as locks) should likely be used to guarantee correctness (other approaches, such as building **lock-free** data structures, are complex and only used on occasion; see the chapter on deadlock in the piece on concurrency for details). For example, assume we have a shared queue being accessed on multiple CPUs concurrently. Without locks, adding or removing elements from the queue concurrently will not work as expected, even with the underlying coherence protocols; one needs locks to atomically update the data structure to its new state.

To make this more concrete, imagine this code sequence, which is used to remove an element from a shared linked list, as we see in Figure 10.3. Imagine if threads on two CPUs enter this routine at the same time. If Thread 1 executes the first line, it will have the current value of `head` stored in its `tmp` variable; if Thread 2 then executes the first line as well, it also will have the same value of `head` stored in its own private `tmp` variable (`tmp` is allocated on the stack, and thus each thread will have its own private storage for it). Thus, instead of each thread removing an element from the head of the list, each thread will try to remove the

```
1  typedef struct __Node_t {
2      int              value;
3      struct __Node_t *next;
4  } Node_t;
5
6  int List_Pop() {
7      Node_t *tmp = head;         // remember old head ...
8      int value   = head->value;  // ... and its value
9      head        = head->next;   // advance head to next pointer
10     free(tmp);                  // free old head
11     return value;               // return value at head
12 }
```

Figure 10.3: **Simple List Delete Code**

same head element, leading to all sorts of problems (such as an attempted double free of the head element at line 4, as well as potentially returning the same data value twice).

The solution, of course, is to make such routines correct via **locking**. In this case, allocating a simple mutex (e.g., `pthread_mutex_t m;`) and then adding a `lock(&m)` at the beginning of the routine and an `unlock(&m)` at the end will solve the problem, ensuring that the code will execute as desired. Unfortunately, as we will see, such an approach is not without problems, in particular with regards to performance. Specifically, as the number of CPUs grows, access to a synchronized shared data structure becomes quite slow.

10.3 One Final Issue: Cache Affinity

One final issue arises in building a multiprocessor cache scheduler, known as **cache affinity**. This notion is simple: a process, when run on a particular CPU, builds up a fair bit of state in the caches (and TLBs) of the CPU. The next time the process runs, it is often advantageous to run it on the same CPU, as it will run faster if some of its state is already present in the caches on that CPU. If, instead, one runs a process on a different CPU each time, the performance of the process will be worse, as it will have to reload the state each time it runs (note it will run correctly on a different CPU thanks to the cache coherence protocols of the hardware). Thus, a multiprocessor scheduler should consider cache affinity when making its scheduling decisions, perhaps preferring to keep a process on the same CPU if at all possible.

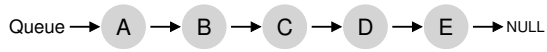
10.4 Single-Queue Scheduling

With this background in place, we now discuss how to build a scheduler for a multiprocessor system. The most basic approach is to simply reuse the basic framework for single processor scheduling, by putting all jobs that need to be scheduled into a single queue; we call this **single-queue multiprocessor scheduling** or **SQMS** for short. This approach has the advantage of simplicity; it does not require much work to take an existing policy that picks the best job to run next and adapt it to work on more than one CPU (where it might pick the best two jobs to run, if there are two CPUs, for example).

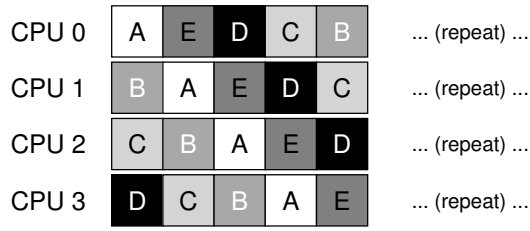
However, SQMS has obvious shortcomings. The first problem is a lack of **scalability**. To ensure the scheduler works correctly on multiple CPUs, the developers will have inserted some form of **locking** into the code, as described above. Locks ensure that when SQMS code accesses the single queue (say, to find the next job to run), the proper outcome arises.

Locks, unfortunately, can greatly reduce performance, particularly as the number of CPUs in the systems grows [A91]. As contention for such a single lock increases, the system spends more and more time in lock overhead and less time doing the work the system should be doing (note: it would be great to include a real measurement of this in here someday).

The second main problem with SQMS is cache affinity. For example, let us assume we have five jobs to run (A, B, C, D, E) and four processors. Our scheduling queue thus looks like this:

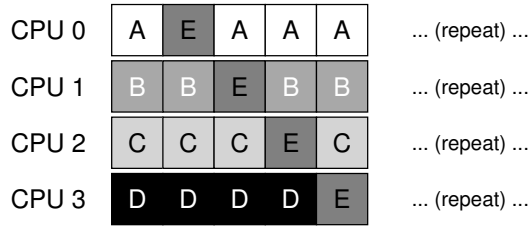


Over time, assuming each job runs for a time slice and then another job is chosen, here is a possible job schedule across CPUs:



Because each CPU simply picks the next job to run from the globally-shared queue, each job ends up bouncing around from CPU to CPU, thus doing exactly the opposite of what would make sense from the standpoint of cache affinity.

To handle this problem, most SQMS schedulers include some kind of affinity mechanism to try to make it more likely that process will continue to run on the same CPU if possible. Specifically, one might provide affinity for some jobs, but move others around to balance load. For example, imagine the same five jobs scheduled as follows:



In this arrangement, jobs A through D are not moved across processors, with only job E **migrating** from CPU to CPU, thus preserving affinity for most. You could then decide to migrate a different job the next time through, thus achieving some kind of affinity fairness as well. Implementing such a scheme, however, can be complex.

Thus, we can see the SQMS approach has its strengths and weaknesses. It is straightforward to implement given an existing single-CPU scheduler, which by definition has only a single queue. However, it does not scale well (due to synchronization overheads), and it does not readily preserve cache affinity.

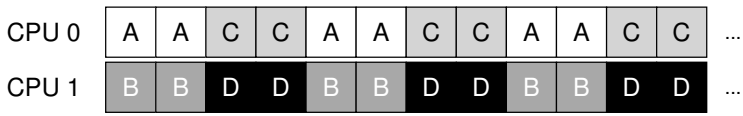
10.5 Multi-Queue Scheduling

Because of the problems caused in single-queue schedulers, some systems opt for multiple queues, e.g., one per CPU. We call this approach **multi-queue multiprocessor scheduling** (or **MQMS**).

In MQMS, our basic scheduling framework consists of multiple scheduling queues. Each queue will likely follow a particular scheduling discipline, such as round robin, though of course any algorithm can be used. When a job enters the system, it is placed on exactly one scheduling queue, according to some heuristic (e.g., random, or picking one with fewer jobs than others). Then it is scheduled essentially independently, thus avoiding the problems of information sharing and synchronization found in the single-queue approach.

For example, assume we have a system where there are just two CPUs (labeled CPU 0 and CPU 1), and some number of jobs enter the system: A, B, C, and D for example. Given that each CPU has a scheduling queue now, the OS has to decide into which queue to place each job. It might do something like this:

Depending on the queue scheduling policy, each CPU now has two jobs to choose from when deciding what should run. For example, with **round robin**, the system might produce a schedule that looks like this:



MQMS has a distinct advantage of SQMS in that it should be inherently more scalable. As the number of CPUs grows, so too does the number of queues, and thus lock and cache contention should not become a central problem. In addition, MQMS intrinsically provides cache affinity;

jobs stay on the same CPU and thus reap the advantage of reusing cached contents therein.

But, if you've been paying attention, you might see that we have a new problem, which is fundamental in the multi-queue based approach: **load imbalance**. Let's assume we have the same set up as above (four jobs, two CPUs), but then one of the jobs (say C) finishes. We now have the following scheduling queues:

If we then run our round-robin policy on each queue of the system, we will see this resulting schedule:

As you can see from this diagram, A gets twice as much CPU as B and D, which is not the desired outcome. Even worse, let's imagine that both A and C finish, leaving just jobs B and D in the system. The scheduling queues will look like this:

As a result, CPU 0 will be left idle! *(insert dramatic and sinister music here)* And hence our CPU usage timeline looks sad:

So what should a poor multi-queue multiprocessor scheduler do? How can we overcome the insidious problem of load imbalance and defeat the evil forces of ... the Decepticons[1]? How do we stop asking questions that are hardly relevant to this otherwise wonderful book?

[1]Little known fact is that the home planet of Cybertron was destroyed by bad CPU scheduling decisions. And now let that be the first and last reference to Transformers in this book, for which we sincerely apologize.

CRUX: HOW TO DEAL WITH LOAD IMBALANCE
How should a multi-queue multiprocessor scheduler handle load imbalance, so as to better achieve its desired scheduling goals?

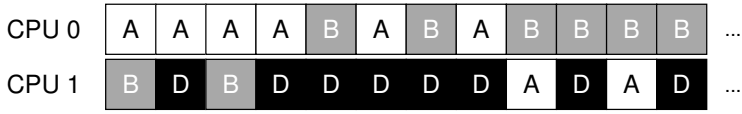
The obvious answer to this query is to move jobs around, a technique which we (once again) refer to as **migration**. By migrating a job from one CPU to another, true load balance can be achieved.

Let's look at a couple of examples to add some clarity. Once again, we have a situation where one CPU is idle and the other has some jobs.

In this case, the desired migration is easy to understand: the OS should simply move one of B or D to CPU 0. The result of this single job migration is evenly balanced load and everyone is happy.

A more tricky case arises in our earlier example, where A was left alone on CPU 0 and B and D were alternating on CPU 1:

In this case, a single migration does not solve the problem. What would you do in this case? The answer, alas, is continuous migration of one or more jobs. One possible solution is to keep switching jobs, as we see in the following timeline. In the figure, first A is alone on CPU 0, and B and D alternate on CPU 1. After a few time slices, B is moved to compete with A on CPU 0, while D enjoys a few time slices alone on CPU 1. And thus load is balanced:



Of course, many other possible migration patterns exist. But now for the tricky part: how should the system decide to enact such a migration?

One basic approach is to use a technique known as **work stealing** [FLR98]. With a work-stealing approach, a (source) queue that is low on jobs will occasionally peek at another (target) queue, to see how full it is. If the target queue is (notably) more full than the source queue, the source will "steal" one or more jobs from the target to help balance load.

Of course, there is a natural tension in such an approach. If you look around at other queues too often, you will suffer from high overhead and have trouble scaling, which was the entire purpose of implementing

the multiple queue scheduling in the first place! If, on the other hand, you don't look at other queues very often, you are in danger of suffering from severe load imbalances. Finding the right threshold remains, as is common in system policy design, a black art.

10.6 Linux Multiprocessor Schedulers

Interestingly, in the Linux community, no common solution has approached to building a multiprocessor scheduler. Over time, three different schedulers arose: the O(1) scheduler, the Completely Fair Scheduler (CFS), and the BF Scheduler (BFS)[2]. See Meehean's dissertation for an excellent overview of the strengths and weaknesses of said schedulers [M11]; here we just summarize a few of the basics.

Both O(1) and CFS use multiple queues, whereas BFS uses a single queue, showing that both approaches can be successful. Of course, there are many other details which separate these schedulers. For example, the O(1) scheduler is a priority-based scheduler (similar to the MLFQ discussed before), changing a process's priority over time and then scheduling those with highest priority in order to meet various scheduling objectives; interactivity is a particular focus. CFS, in contrast, is a deterministic proportional-share approach (more like Stride scheduling, as discussed earlier). BFS, the only single-queue approach among the three, is also proportional-share, but based on a more complicated scheme known as Earliest Eligible Virtual Deadline First (EEVDF) [SA96]. Read more about these modern algorithms on your own; you should be able to understand how they work now!

10.7 Summary

We have seen various approaches to multiprocessor scheduling. The single-queue approach (SQMS) is rather straightforward to build and balances load well but inherently has difficulty with scaling to many processors and cache affinity. The multiple-queue approach (MQMS) scales better and handles cache affinity well, but has trouble with load imbalance and is more complicated. Whichever approach you take, there is no simple answer: building a general purpose scheduler remains a daunting task, as small code changes can lead to large behavioral differences. Only undertake such an exercise if you know exactly what you are doing, or, at least, are getting paid a large amount of money to do so.

[2] Look up what BF stands for on your own; be forewarned, it is not for the faint of heart.

References

[A90] "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors"
Thomas E. Anderson
IEEE TPDS Volume 1:1, January 1990
A classic paper on how different locking alternatives do and don't scale. By Tom Anderson, very well known researcher in both systems and networking. And author of a very fine OS textbook, we must say.

[B+10] "An Analysis of Linux Scalability to Many Cores Abstract"
Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, Nickolai Zeldovich
OSDI '10, Vancouver, Canada, October 2010
A terrific modern paper on the difficulties of scaling Linux to many cores.

[CSG99] "Parallel Computer Architecture: A Hardware/Software Approach"
David E. Culler, Jaswinder Pal Singh, and Anoop Gupta
Morgan Kaufmann, 1999
A treasure filled with details about parallel machines and algorithms. As Mark Hill humorously observes on the jacket, the book contains more information than most research papers.

[FLR98] "The Implementation of the Cilk-5 Multithreaded Language"
Matteo Frigo, Charles E. Leiserson, Keith Randall
PLDI '98, Montreal, Canada, June 1998
Cilk is a lightweight language and runtime for writing parallel programs, and an excellent example of the work-stealing paradigm.

[G83] "Using Cache Memory To Reduce Processor-Memory Traffic"
James R. Goodman
ISCA '83, Stockholm, Sweden, June 1983
The pioneering paper on how to use bus snooping, i.e., paying attention to requests you see on the bus, to build a cache coherence protocol. Goodman's research over many years at Wisconsin is full of cleverness, this being but one example.

[M11] "Towards Transparent CPU Scheduling"
Joseph T. Meehean
Doctoral Dissertation at University of Wisconsin—Madison, 2011
A dissertation that covers a lot of the details of how modern Linux multiprocessor scheduling works. Pretty awesome! But, as co-advisors of Joe's, we may be a bit biased here.

[SHW11] "A Primer on Memory Consistency and Cache Coherence"
Daniel J. Sorin, Mark D. Hill, and David A. Wood
Synthesis Lectures in Computer Architecture
Morgan and Claypool Publishers, May 2011
A definitive overview of memory consistency and multiprocessor caching. Required reading for anyone who likes to know way too much about a given topic.

[SA96] "Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation"
Ion Stoica and Hussein Abdel-Wahab
Technical Report TR-95-22, Old Dominion University, 1996
A tech report on this cool scheduling idea, from Ion Stoica, now a professor at U.C. Berkeley and world expert in networking, distributed systems, and many other things.

11

Summary Dialogue on CPU Virtualization

Professor: *So, Student, did you learn anything?*

Student: *Well, Professor, that seems like a loaded question. I think you only want me to say "yes."*

Professor: *That's true. But it's also still an honest question. Come on, give a professor a break, will you?*

Student: *OK, OK. I think I did learn a few things. First, I learned a little about how the OS virtualizes the CPU. There are a bunch of important* ***mechanisms*** *that I had to understand to make sense of this: traps and trap handlers, timer interrupts, and how the OS and the hardware have to carefully save and restore state when switching between processes.*

Professor: *Good, good!*

Student: *All those interactions do seem a little complicated though; how can I learn more?*

Professor: *Well, that's a good question. I think there is no substitute for doing; just reading about these things doesn't quite give you the proper sense. Do the class projects and I bet by the end it will all kind of make sense.*

Student: *Sounds good. What else can I tell you?*

Professor: *Well, did you get some sense of the philosophy of the OS in your quest to understand its basic machinery?*

Student: *Hmm... I think so. It seems like the OS is fairly paranoid. It wants to make sure it stays in charge of the machine. While it wants a program to run as efficiently as possible (and hence the whole reasoning behind* ***limited direct execution****), the OS also wants to be able to say "Ah! Not so fast my friend" in case of an errant or malicious process. Paranoia rules the day, and certainly keeps the OS in charge of the machine. Perhaps that is why we think of the OS as a resource manager.*

Professor: *Yes indeed — sounds like you are starting to put it together! Nice.*

Student: *Thanks.*

Professor: *And what about the policies on top of those mechanisms — any interesting lessons there?*

Student: *Some lessons to be learned there for sure. Perhaps a little obvious, but obvious can be good. Like the notion of bumping short jobs to the front of the queue — I knew that was a good idea ever since the one time I was buying some gum at the store, and the guy in front of me had a credit card that wouldn't work. He was no short job, let me tell you.*

Professor: *That sounds oddly rude to that poor fellow. What else?*

Student: *Well, that you can build a smart scheduler that tries to be like SJF and RR all at once — that MLFQ was pretty neat. Building up a real scheduler seems difficult.*

Professor: *Indeed it is. That's why there is still controversy to this day over which scheduler to use; see the Linux battles between CFS, BFS, and the O(1) scheduler, for example. And no, I will not spell out the full name of BFS.*

Student: *And I won't ask you to! These policy battles seem like they could rage forever; is there really a right answer?*

Professor: *Probably not. After all, even our own metrics are at odds: if your scheduler is good at turnaround time, it's bad at response time, and vice versa. As Lampson said, perhaps the goal isn't to find the best solution, but rather to avoid disaster.*

Student: *That's a little depressing.*

Professor: *Good engineering can be that way. And it can also be uplifting! It's just your perspective on it, really. I personally think being pragmatic is a good thing, and pragmatists realize that not all problems have clean and easy solutions. Anything else that caught your fancy?*

Student: *I really liked the notion of gaming the scheduler; it seems like that might be something to look into when I'm next running a job on Amazon's EC2 service. Maybe I can steal some cycles from some other unsuspecting (and more importantly, OS-ignorant) customer!*

Professor: *It looks like I might have created a monster! Professor Frankenstein is not what I'd like to be called, you know.*

Student: *But isn't that the idea? To get us excited about something, so much so that we look into it on our own? Lighting fires and all that?*

Professor: *I guess so. But I didn't think it would work!*

12

A Dialogue on Memory Virtualization

Student: *So, are we done with virtualization?*

Professor: *No!*

Student: *Hey, no reason to get so excited; I was just asking a question. Students are supposed to do that, right?*

Professor: *Well, professors do always say that, but really they mean this: ask questions, **if** they are good questions, **and** you have actually put a little thought into them.*

Student: *Well, that sure takes the wind out of my sails.*

Professor: *Mission accomplished. In any case, we are not nearly done with virtualization! Rather, you have just seen how to virtualize the CPU, but really there is a big monster waiting in the closet: memory. Virtualizing memory is complicated and requires us to understand many more intricate details about how the hardware and OS interact.*

Student: *That sounds cool. Why is it so hard?*

Professor: *Well, there are a lot of details, and you have to keep them straight in your head to really develop a mental model of what is going on. We'll start simple, with very basic techniques like base/bounds, and slowly add complexity to tackle new challenges, including fun topics like TLBs and multi-level page tables. Eventually, we'll be able to describe the workings of a fully-functional modern virtual memory manager.*

Student: *Neat! Any tips for the poor student, inundated with all of this information and generally sleep-deprived?*

Professor: *For the sleep deprivation, that's easy: sleep more (and party less). For understanding virtual memory, start with this: **every address generated by a user program is a virtual address**. The OS is just providing an illusion to each process, specifically that it has its own large and private memory; with some hardware help, the OS will turn these pretend virtual addresses into real physical addresses, and thus be able to locate the desired information.*

Student: *OK, I think I can remember that... (to self) every address from a user program is virtual, every address from a user program is virtual, every ...*

Professor: *What are you mumbling about?*

Student: *Oh nothing.... (awkward pause) ... Anyway, why does the OS want to provide this illusion again?*

Professor: *Mostly **ease of use**: the OS will give each program the view that it has a large contiguous **address space** to put its code and data into; thus, as a programmer, you never have to worry about things like "where should I store this variable?" because the virtual address space of the program is large and has lots of room for that sort of thing. Life, for a programmer, becomes much more tricky if you have to worry about fitting all of your code data into a small, crowded memory.*

Student: *Why else?*

Professor: *Well, **isolation** and **protection** are big deals, too. We don't want one errant program to be able to read, or worse, overwrite, some other program's memory, do we?*

Student: *Probably not. Unless it's a program written by someone you don't like.*

Professor: *Hmmm.... I think we might need to add a class on morals and ethics to your schedule for next semester. Perhaps OS class isn't getting the right message across.*

Student: *Maybe we should. But remember, it's not me who taught us that the proper OS response to errant process behavior is to kill the offending process!*

13

The Abstraction: Address Spaces

In the early days, building computer systems was easy. Why, you ask? Because users didn't expect much. It is those darned users with their expectations of "ease of use", "high performance", "reliability", etc., that really have led to all these headaches. Next time you meet one of those computer users, thank them for all the problems they have caused.

13.1 Early Systems

From the perspective of memory, early machines didn't provide much of an abstraction to users. Basically, the physical memory of the machine looked something like what you see in Figure 13.1.

The OS was a set of routines (a library, really) that sat in memory (starting at physical address 0 in this example), and there would be one running program (a process) that currently sat in physical memory (starting at physical address 64k in this example) and used the rest of memory. There were few illusions here, and the user didn't expect much from the OS. Life was sure easy for OS developers in those days, wasn't it?

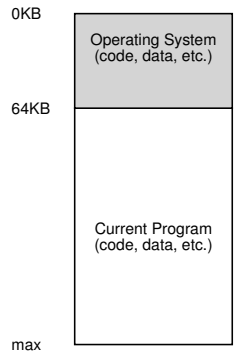


Figure 13.1: **Operating Systems: The Early Days**

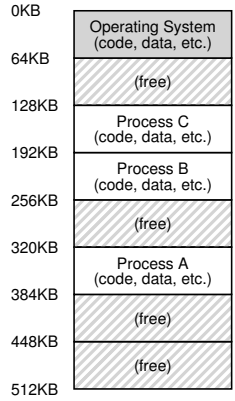


Figure 13.2: **Three Processes: Sharing Memory**

13.2 Multiprogramming and Time Sharing

After a time, because machines were expensive, people began to share machines more effectively. Thus the era of **multiprogramming** was born [DV66], in which multiple processes were ready to run at a given time, and the OS would switch between them, for example when one decided to perform an I/O. Doing so increased the effective **utilization** of the CPU. Such increases in **efficiency** were particularly important in those days where each machine cost hundreds of thousands or even millions of dollars (and you thought your Mac was expensive!).

Soon enough, however, people began demanding more of machines, and the era of **time sharing** was born [S59, L60, M62, M83]. Specifically, many realized the limitations of batch computing, particularly on programmers themselves [CV65], who were tired of long (and hence ineffective) program-debug cycles. The notion of **interactivity** became important, as many users might be concurrently using a machine, each waiting for (or hoping for) a timely response from their currently-executing tasks.

One way to implement time sharing would be to run one process for a short while, giving it full access to all memory (Figure 13.1, page 1), then stop it, save all of its state to some kind of disk (including all of physical memory), load some other process's state, run it for a while, and thus implement some kind of crude sharing of the machine [M+63].

Unfortunately, this approach has a big problem: it is way too slow, particularly as memory grows. While saving and restoring register-level state (the PC, general-purpose registers, etc.) is relatively fast, saving the entire contents of memory to disk is brutally non-performant. Thus, what we'd rather do is leave processes in memory while switching between them, allowing the OS to implement time sharing efficiently (Figure 13.2).

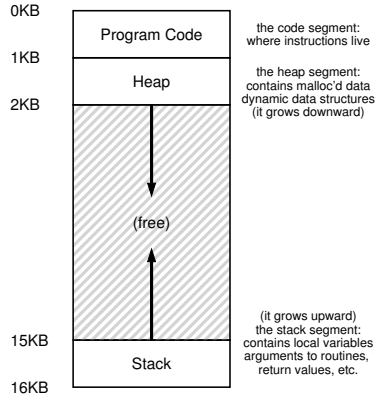


Figure 13.3: **An Example Address Space**

In the diagram, there are three processes (A, B, and C) and each of them have a small part of the 512KB physical memory carved out for them. Assuming a single CPU, the OS chooses to run one of the processes (say A), while the others (B and C) sit in the ready queue waiting to run.

As time sharing became more popular, you can probably guess that new demands were placed on the operating system. In particular, allowing multiple programs to reside concurrently in memory makes **protection** an important issue; you don't want a process to be able to read, or worse, write some other process's memory.

13.3 The Address Space

However, we have to keep those pesky users in mind, and doing so requires the OS to create an **easy to use** abstraction of physical memory. We call this abstraction the **address space**, and it is the running program's view of memory in the system. Understanding this fundamental OS abstraction of memory is key to understanding how memory is virtualized.

The address space of a process contains all of the memory state of the running program. For example, the **code** of the program (the instructions) have to live in memory somewhere, and thus they are in the address space. The program, while it is running, uses a **stack** to keep track of where it is in the function call chain as well as to allocate local variables and pass parameters and return values to and from routines. Finally, the **heap** is used for dynamically-allocated, user-managed memory, such as that you might receive from a call to `malloc()` in C or `new` in an object-oriented language such as C++ or Java. Of course, there are other things in there too (e.g., statically-initialized variables), but for now let us just assume those three components: code, stack, and heap.

In the example in Figure 13.3 (page 3), we have a tiny address space (only 16KB)[1]. The program code lives at the top of the address space (starting at 0 in this example, and is packed into the first 1K of the address space). Code is static (and thus easy to place in memory), so we can place it at the top of the address space and know that it won't need any more space as the program runs.

Next, we have the two regions of the address space that may grow (and shrink) while the program runs. Those are the heap (at the top) and the stack (at the bottom). We place them like this because each wishes to be able to grow, and by putting them at opposite ends of the address space, we can allow such growth: they just have to grow in opposite directions. The heap thus starts just after the code (at 1KB) and grows downward (say when a user requests more memory via `malloc()`); the stack starts at 16KB and grows upward (say when a user makes a procedure call). However, this placement of stack and heap is just a convention; you could arrange the address space in a different way if you'd like (as we'll see later, when multiple **threads** co-exist in an address space, no nice way to divide the address space like this works anymore, alas).

Of course, when we describe the address space, what we are describing is the **abstraction** that the OS is providing to the running program. The program really isn't in memory at physical addresses 0 through 16KB; rather it is loaded at some arbitrary physical address(es). Examine processes A, B, and C in Figure 13.2; there you can see how each process is loaded into memory at a different address. And hence the problem:

THE CRUX: HOW TO VIRTUALIZE MEMORY

How can the OS build this abstraction of a private, potentially large address space for multiple running processes (all sharing memory) on top of a single, physical memory?

When the OS does this, we say the OS is **virtualizing memory**, because the running program thinks it is loaded into memory at a particular address (say 0) and has a potentially very large address space (say 32-bits or 64-bits); the reality is quite different.

When, for example, process A in Figure 13.2 tries to perform a load at address 0 (which we will call a **virtual address**), somehow the OS, in tandem with some hardware support, will have to make sure the load doesn't actually go to physical address 0 but rather to physical address 320KB (where A is loaded into memory). This is the key to virtualization of memory, which underlies every modern computer system in the world.

[1] We will often use small examples like this because (a) it is a pain to represent a 32-bit address space and (b) the math is harder. We like simple math.

TIP: THE PRINCIPLE OF ISOLATION
Isolation is a key principle in building reliable systems. If two entities are properly isolated from one another, this implies that one can fail without affecting the other. Operating systems strive to isolate processes from each other and in this way prevent one from harming the other. By using memory isolation, the OS further ensures that running programs cannot affect the operation of the underlying OS. Some modern OS's take isolation even further, by walling off pieces of the OS from other pieces of the OS. Such **microkernels** [BH70, R+89, S+03] thus may provide greater reliability than typical monolithic kernel designs.

13.4 Goals

Thus we arrive at the job of the OS in this set of notes: to virtualize memory. The OS will not only virtualize memory, though; it will do so with style. To make sure the OS does so, we need some goals to guide us. We have seen these goals before (think of the Introduction), and we'll see them again, but they are certainly worth repeating.

One major goal of a virtual memory (VM) system is **transparency**[2]. The OS should implement virtual memory in a way that is invisible to the running program. Thus, the program shouldn't be aware of the fact that memory is virtualized; rather, the program behaves as if it has its own private physical memory. Behind the scenes, the OS (and hardware) does all the work to multiplex memory among many different jobs, and hence implements the illusion.

Another goal of VM is **efficiency**. The OS should strive to make the virtualization as **efficient** as possible, both in terms of time (i.e., not making programs run much more slowly) and space (i.e., not using too much memory for structures needed to support virtualization). In implementing time-efficient virtualization, the OS will have to rely on hardware support, including hardware features such as TLBs (which we will learn about in due course).

Finally, a third VM goal is **protection**. The OS should make sure to **protect** processes from one another as well as the OS itself from processes. When one process performs a load, a store, or an instruction fetch, it should not be able to access or affect in any way the memory contents of any other process or the OS itself (that is, anything *outside* its address space). Protection thus enables us to deliver the property of **isolation** among processes; each process should be running in its own isolated cocoon, safe from the ravages of other faulty or even malicious processes.

[2]This usage of transparency is sometimes confusing; some students think that "being transparent" means keeping everything out in the open, i.e., what government should be like. Here, it means the opposite: that the illusion provided by the OS should not be visible to applications. Thus, in common usage, a transparent system is one that is hard to notice, not one that responds to requests as stipulated by the Freedom of Information Act.

ASIDE: **EVERY ADDRESS YOU SEE IS VIRTUAL**

Ever write a C program that prints out a pointer? The value you see (some large number, often printed in hexadecimal), is a **virtual address**. Ever wonder where the code of your program is found? You can print that out too, and yes, if you can print it, it also is a virtual address. In fact, any address you can see as a programmer of a user-level program is a virtual address. It's only the OS, through its tricky techniques of virtualizing memory, that knows where in the physical memory of the machine these instructions and data values lie. So never forget: if you print out an address in a program, it's a virtual one, an illusion of how things are laid out in memory; only the OS (and the hardware) knows the real truth.

Here's a little program that prints out the locations of the `main()` routine (where code lives), the value of a heap-allocated value returned from `malloc()`, and the location of an integer on the stack:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(int argc, char *argv[]) {
4     printf("location of code  : %p\n", (void *) main);
5     printf("location of heap  : %p\n", (void *) malloc(1));
6     int x = 3;
7     printf("location of stack : %p\n", (void *) &x);
8     return x;
9 }
```

When run on a 64-bit Mac OS X machine, we get the following output:

```
location of code  : 0x1095afe50
location of heap  : 0x1096008c0
location of stack : 0x7fff691aea64
```

From this, you can see that code comes first in the address space, then the heap, and the stack is all the way at the other end of this large virtual space. All of these addresses are virtual, and will be translated by the OS and hardware in order to fetch values from their true physical locations.

In the next chapters, we'll focus our exploration on the basic **mechanisms** needed to virtualize memory, including hardware and operating systems support. We'll also investigate some of the more relevant **policies** that you'll encounter in operating systems, including how to manage free space and which pages to kick out of memory when you run low on space. In doing so, we'll build up your understanding of how a modern virtual memory system really works[3].

[3] Or, we'll convince you to drop the course. But hold on; if you make it through VM, you'll likely make it all the way!

13.5 Summary

We have seen the introduction of a major OS subsystem: virtual memory. The VM system is responsible for providing the illusion of a large, sparse, private address space to programs, which hold all of their instructions and data therein. The OS, with some serious hardware help, will take each of these virtual memory references, and turn them into physical addresses, which can be presented to the physical memory in order to fetch the desired information. The OS will do this for many processes at once, making sure to protect programs from one another, as well as protect the OS. The entire approach requires a great deal of mechanism (lots of low-level machinery) as well as some critical policies to work; we'll start from the bottom up, describing the critical mechanisms first. And thus we proceed!

References

[BH70] "The Nucleus of a Multiprogramming System"
Per Brinch Hansen
Communications of the ACM, 13:4, April 1970
The first paper to suggest that the OS, or kernel, should be a minimal and flexible substrate for building customized operating systems; this theme is revisited throughout OS research history.

[CV65] "Introduction and Overview of the Multics System"
F. J. Corbato and V. A. Vyssotsky
Fall Joint Computer Conference, 1965
A great early Multics paper. Here is the great quote about time sharing: "The impetus for time-sharing first arose from professional programmers because of their constant frustration in debugging programs at batch processing installations. Thus, the original goal was to time-share computers to allow simultaneous access by several persons while giving to each of them the illusion of having the whole machine at his disposal."

[DV66] "Programming Semantics for Multiprogrammed Computations"
Jack B. Dennis and Earl C. Van Horn
Communications of the ACM, Volume 9, Number 3, March 1966
An early paper (but not the first) on multiprogramming.

[L60] "Man-Computer Symbiosis"
J. C. R. Licklider
IRE Transactions on Human Factors in Electronics, HFE-1:1, March 1960
A funky paper about how computers and people are going to enter into a symbiotic age; clearly well ahead of its time but a fascinating read nonetheless.

[M62] "Time-Sharing Computer Systems"
J. McCarthy
Management and the Computer of the Future, MIT Press, Cambridge, Mass, 1962
Probably McCarthy's earliest recorded paper on time sharing. However, in another paper [M83], he claims to have been thinking of the idea since 1957. McCarthy left the systems area and went on to become a giant in Artificial Intelligence at Stanford, including the creation of the LISP programming language. See McCarthy's home page for more info: `http://www-formal.stanford.edu/jmc/`

[M+63] "A Time-Sharing Debugging System for a Small Computer"
J. McCarthy, S. Boilen, E. Fredkin, J. C. R. Licklider
AFIPS '63 (Spring), May, 1963, New York, USA
A great early example of a system that swapped program memory to the "drum" when the program wasn't running, and then back into "core" memory when it was about to be run.

[M83] "Reminiscences on the History of Time Sharing"
John McCarthy
Winter or Spring of 1983
Available: http://www-formal.stanford.edu/jmc/history/timesharing/timesharing.html
A terrific historical note on where the idea of time-sharing might have come from, including some doubts towards those who cite Strachey's work [S59] as the pioneering work in this area.

[R+89] "Mach: A System Software kernel"
Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alessandro Forin, David Golub, Michael Jones
COMPCON 89, February 1989
Although not the first project on microkernels per se, the Mach project at CMU was well-known and influential; it still lives today deep in the bowels of Mac OS X.

[S59] "Time Sharing in Large Fast Computers"
C. Strachey
Proceedings of the International Conference on Information Processing, UNESCO, June 1959
One of the earliest references on time sharing.

[S+03] "Improving the Reliability of Commodity Operating Systems"
Michael M. Swift, Brian N. Bershad, Henry M. Levy
SOSP 2003
The first paper to show how microkernel-like thinking can improve operating system reliability.

14

Interlude: Memory API

In this interlude, we discuss the memory allocation interfaces in UNIX systems. The interfaces provided are quite simple, and hence the chapter is short and to the point[1]. The main problem we address is this:

CRUX: HOW TO ALLOCATE AND MANAGE MEMORY
In UNIX/C programs, understanding how to allocate and manage memory is critical in building robust and reliable software. What interfaces are commonly used? What mistakes should be avoided?

14.1 Types of Memory

In running a C program, there are two types of memory that are allocated. The first is called **stack** memory, and allocations and deallocations of it are managed *implicitly* by the compiler for you, the programmer; for this reason it is sometimes called **automatic** memory.

Declaring memory on the stack in C is easy. For example, let's say you need some space in a function `func()` for an integer, called `x`. To declare such a piece of memory, you just do something like this:

```
void func() {
    int x; // declares an integer on the stack
    ...
}
```

The compiler does the rest, making sure to make space on the stack when you call into `func()`. When you return from the function, the compiler deallocates the memory for you; thus, if you want some information to live beyond the call invocation, you had better not leave that information on the stack.

It is this need for long-lived memory that gets us to the second type of memory, called **heap** memory, where all allocations and deallocations

[1] Indeed, we hope all chapters are! But this one is shorter and pointier, we think.

are *explicitly* handled by you, the programmer. A heavy responsibility, no doubt! And certainly the cause of many bugs. But if you are careful and pay attention, you will use such interfaces correctly and without too much trouble. Here is an example of how one might allocate a pointer to an integer on the heap:

```
void func() {
    int *x = (int *) malloc(sizeof(int));
    ...
}
```

A couple of notes about this small code snippet. First, you might notice that both stack and heap allocation occur on this line: first the compiler knows to make room for a pointer to an integer when it sees your declaration of said pointer (`int *x`); subsequently, when the program calls `malloc()`, it requests space for an integer on the heap; the routine returns the address of such an integer (upon success, or `NULL` on failure), which is then stored on the stack for use by the program.

Because of its explicit nature, and because of its more varied usage, heap memory presents more challenges to both users and systems. Thus, it is the focus of the remainder of our discussion.

14.2 The `malloc()` Call

The `malloc()` call is quite simple: you pass it a size asking for some room on the heap, and it either succeeds and gives you back a pointer to the newly-allocated space, or fails and returns `NULL`[2].

The manual page shows what you need to do to use malloc; type `man malloc` at the command line and you will see:

```
#include <stdlib.h>
...
void *malloc(size_t size);
```

From this information, you can see that all you need to do is include the header file `stdlib.h` to use malloc. In fact, you don't really need to even do this, as the C library, which all C programs link with by default, has the code for `malloc()` inside of it; adding the header just lets the compiler check whether you are calling `malloc()` correctly (e.g., passing the right number of arguments to it, of the right type).

The single parameter `malloc()` takes is of type `size_t` which simply describes how many bytes you need. However, most programmers do not type in a number here directly (such as 10); indeed, it would be considered poor form to do so. Instead, various routines and macros are utilized. For example, to allocate space for a double-precision floating point value, you simply do this:

```
double *d = (double *) malloc(sizeof(double));
```

[2] Note that `NULL` in C isn't really anything special at all, just a macro for the value zero.

TIP: WHEN IN DOUBT, TRY IT OUT

If you aren't sure how some routine or operator you are using behaves, there is no substitute for simply trying it out and making sure it behaves as you expect. While reading the manual pages or other documentation is useful, how it works in practice is what matters. Write some code and test it! That is no doubt the best way to make sure your code behaves as you desire. Indeed, that is what we did to double-check the things we were saying about `sizeof()` were actually true!

Wow, that's lot of `double`-ing! This invocation of `malloc()` uses the `sizeof()` operator to request the right amount of space; in C, this is generally thought of as a *compile-time* operator, meaning that the actual size is known at *compile time* and thus a number (in this case, 8, for a double) is substituted as the argument to `malloc()`. For this reason, `sizeof()` is correctly thought of as an operator and not a function call (a function call would take place at run time).

You can also pass in the name of a variable (and not just a type) to `sizeof()`, but in some cases you may not get the desired results, so be careful. For example, let's look at the following code snippet:

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));
```

In the first line, we've declared space for an array of 10 integers, which is fine and dandy. However, when we use `sizeof()` in the next line, it returns a small value, such as 4 (on 32-bit machines) or 8 (on 64-bit machines). The reason is that in this case, `sizeof()` thinks we are simply asking how big a *pointer* to an integer is, not how much memory we have dynamically allocated. However, sometimes `sizeof()` does work as you might expect:

```
int x[10];
printf("%d\n", sizeof(x));
```

In this case, there is enough static information for the compiler to know that 40 bytes have been allocated.

Another place to be careful is with strings. When declaring space for a string, use the following idiom: `malloc(strlen(s) + 1)`, which gets the length of the string using the function `strlen()`, and adds 1 to it in order to make room for the end-of-string character. Using `sizeof()` may lead to trouble here.

You might also notice that `malloc()` returns a pointer to type `void`. Doing so is just the way in C to pass back an address and let the programmer decide what to do with it. The programmer further helps out by using what is called a **cast**; in our example above, the programmer casts the return type of `malloc()` to a pointer to a `double`. Casting doesn't really accomplish anything, other than tell the compiler and other

programmers who might be reading your code: "yeah, I know what I'm doing." By casting the result of `malloc()`, the programmer is just giving some reassurance; the cast is not needed for the correctness.

14.3 The `free()` Call

As it turns out, allocating memory is the easy part of the equation; knowing when, how, and even if to free memory is the hard part. To free heap memory that is no longer in use, programmers simply call `free()`:

```
int *x = malloc(10 * sizeof(int));
...
free(x);
```

The routine takes one argument, a pointer that was returned by `malloc()`. Thus, you might notice, the size of the allocated region is not passed in by the user, and must be tracked by the memory-allocation library itself.

14.4 Common Errors

There are a number of common errors that arise in the use of `malloc()` and `free()`. Here are some we've seen over and over again in teaching the undergraduate operating systems course. All of these examples compile and run with nary a peep from the compiler; while compiling a C program is necessary to build a correct C program, it is far from sufficient, as you will learn (often in the hard way).

Correct memory management has been such a problem, in fact, that many newer languages have support for **automatic memory management**. In such languages, while you call something akin to `malloc()` to allocate memory (usually **new** or something similar to allocate a new object), you never have to call something to free space; rather, a **garbage collector** runs and figures out what memory you no longer have references to and frees it for you.

Forgetting To Allocate Memory

Many routines expect memory to be allocated before you call them. For example, the routine `strcpy(dst, src)` copies a string from a source pointer to a destination pointer. However, if you are not careful, you might do this:

```
char *src = "hello";
char *dst;        // oops! unallocated
strcpy(dst, src); // segfault and die
```

When you run this code, it will likely lead to a **segmentation fault**[3], which is a fancy term for **YOU DID SOMETHING WRONG WITH MEMORY YOU FOOLISH PROGRAMMER AND I AM ANGRY.**

[3] Although it sounds arcane, you will soon learn why such an illegal memory access is called a segmentation fault; if that isn't incentive to read on, what is?

TIP: IT COMPILED OR IT RAN $\neq$ IT IS CORRECT
Just because a program compiled(!) or even ran once or many times correctly does not mean the program is correct. Many events may have conspired to get you to a point where you believe it works, but then something changes and it stops. A common student reaction is to say (or yell) "But it worked before!" and then blame the compiler, operating system, hardware, or even (dare we say it) the professor. But the problem is usually right where you think it would be, in your code. Get to work and debug it before you blame those other components.

In this case, the proper code might instead look like this:

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src) + 1);
strcpy(dst, src); // work properly
```

Alternately, you could use `strdup()` and make your life even easier. Read the `strdup` man page for more information.

Not Allocating Enough Memory

A related error is not allocating enough memory, sometimes called a **buffer overflow**. In the example above, a common error is to make *almost* enough room for the destination buffer.

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src)); // too small!
strcpy(dst, src); // work properly
```

Oddly enough, depending on how malloc is implemented and many other details, this program will often run seemingly correctly. In some cases, when the string copy executes, it writes one byte too far past the end of the allocated space, but in some cases this is harmless, perhaps overwriting a variable that isn't used anymore. In some cases, these overflows can be incredibly harmful, and in fact are the source of many security vulnerabilities in systems [W06]. In other cases, the malloc library allocated a little extra space anyhow, and thus your program actually doesn't scribble on some other variable's value and works quite fine. In even other cases, the program will indeed fault and crash. And thus we learn another valuable lesson: even though it ran correctly once, doesn't mean it's correct.

Forgetting to Initialize Allocated Memory

With this error, you call `malloc()` properly, but forget to fill in some values into your newly-allocated data type. Don't do this! If you do forget, your program will eventually encounter an **uninitialized read**, where it

reads from the heap some data of unknown value. Who knows what might be in there? If you're lucky, some value such that the program still works (e.g., zero). If you're not lucky, something random and harmful.

Forgetting To Free Memory

Another common error is known as a **memory leak**, and it occurs when you forget to free memory. In long-running applications or systems (such as the OS itself), this is a huge problem, as slowly leaking memory eventually leads one to run out of memory, at which point a restart is required. Thus, in general, when you are done with a chunk of memory, you should make sure to free it. Note that using a garbage-collected language doesn't help here: if you still have a reference to some chunk of memory, no garbage collector will ever free it, and thus memory leaks remain a problem even in more modern languages.

In some cases, it may seem like not calling `free()` is reasonable. For example, your program is short-lived, and will soon exit; in this case, when the process dies, the OS will clean up all of its allocated pages and thus no memory leak will take place per se. While this certainly "works" (see the aside on page 7), it is probably a bad habit to develop, so be wary of choosing such a strategy. In the long run, one of your goals as a programmer is to develop good habits; one of those habits is understanding how you are managing memory, and (in languages like C), freeing the blocks you have allocated. Even if you can get away with not doing so, it is probably good to get in the habit of freeing each and every byte you explicitly allocate.

Freeing Memory Before You Are Done With It

Sometimes a program will free memory before it is finished using it; such a mistake is called a **dangling pointer**, and it, as you can guess, is also a bad thing. The subsequent use can crash the program, or overwrite valid memory (e.g., you called `free()`, but then called `malloc()` again to allocate something else, which then recycles the errantly-freed memory).

Freeing Memory Repeatedly

Programs also sometimes free memory more than once; this is known as the **double free**. The result of doing so is undefined. As you can imagine, the memory-allocation library might get confused and do all sorts of weird things; crashes are a common outcome.

Calling `free()` Incorrectly

One last problem we discuss is the call of `free()` incorrectly. After all, `free()` expects you only to pass to it one of the pointers you received from `malloc()` earlier. When you pass in some other value, bad things can (and do) happen. Thus, such **invalid frees** are dangerous and of course should also be avoided.

ASIDE: **WHY NO MEMORY IS LEAKED ONCE YOUR PROCESS EXITS**

When you write a short-lived program, you might allocate some space using `malloc()`. The program runs and is about to complete: is there need to call `free()` a bunch of times just before exiting? While it seems wrong not to, no memory will be "lost" in any real sense. The reason is simple: there are really two levels of memory management in the system.

The first level of memory management is performed by the OS, which hands out memory to processes when they run, and takes it back when processes exit (or otherwise die). The second level of management is *within* each process, for example within the heap when you call `malloc()` and `free()`. Even if you fail to call `free()` (and thus leak memory in the heap), the operating system will reclaim *all* the memory of the process (including those pages for code, stack, and, as relevant here, heap) when the program is finished running. No matter what the state of your heap in your address space, the OS takes back all of those pages when the process dies, thus ensuring that no memory is lost despite the fact that you didn't free it.

Thus, for short-lived programs, leaking memory often does not cause any operational problems (though it may be considered poor form). When you write a long-running server (such as a web server or database management system, which never exit), leaked memory is a much bigger issue, and will eventually lead to a crash when the application runs out of memory. And of course, leaking memory is an even larger issue inside one particular program: the operating system itself. Showing us once again: those who write the kernel code have the toughest job of all...

Summary

As you can see, there are lots of ways to abuse memory. Because of frequent errors with memory, a whole ecosphere of tools have developed to help find such problems in your code. Check out both **purify** [HJ92] and **valgrind** [SN05]; both are excellent at helping you locate the source of your memory-related problems. Once you become accustomed to using these powerful tools, you will wonder how you survived without them.

14.5 Underlying OS Support

You might have noticed that we haven't been talking about system calls when discussing `malloc()` and `free()`. The reason for this is simple: they are not system calls, but rather library calls. Thus the malloc library manages space within your virtual address space, but itself is built on top of some system calls which call into the OS to ask for more memory or release some back to the system.

One such system call is called `brk`, which is used to change the location of the program's **break**: the location of the end of the heap. It takes one argument (the address of the new break), and thus either increases or decreases the size of the heap based on whether the new break is larger or smaller than the current break. An additional call `sbrk` is passed an increment but otherwise serves a similar purpose.

Note that you should never directly call either `brk` or `sbrk`. They are used by the memory-allocation library; if you try to use them, you will likely make something go (horribly) wrong. Stick to `malloc()` and `free()` instead.

Finally, you can also obtain memory from the operating system via the `mmap()` call. By passing in the correct arguments, `mmap()` can create an **anonymous** memory region within your program — a region which is not associated with any particular file but rather with **swap space**, something we'll discuss in detail later on in virtual memory. This memory can then also be treated like a heap and managed as such. Read the manual page of `mmap()` for more details.

14.6 Other Calls

There are a few other calls that the memory-allocation library supports. For example, `calloc()` allocates memory and also zeroes it before returning; this prevents some errors where you assume that memory is zeroed and forget to initialize it yourself (see the paragraph on "uninitialized reads" above). The routine `realloc()` can also be useful, when you've allocated space for something (say, an array), and then need to add something to it: `realloc()` makes a new larger region of memory, copies the old region into it, and returns the pointer to the new region.

14.7 Summary

We have introduced some of the APIs dealing with memory allocation. As always, we have just covered the basics; more details are available elsewhere. Read the C book [KR88] and Stevens [SR05] (Chapter 7) for more information. For a cool modern paper on how to detect and correct many of these problems automatically, see Novark et al. [N+07]; this paper also contains a nice summary of common problems and some neat ideas on how to find and fix them.

References

[HJ92] Purify: Fast Detection of Memory Leaks and Access Errors
R. Hastings and B. Joyce
USENIX Winter '92
The paper behind the cool Purify tool, now a commercial product.

[KR88] "The C Programming Language"
Brian Kernighan and Dennis Ritchie
Prentice-Hall 1988
The C book, by the developers of C. Read it once, do some programming, then read it again, and then keep it near your desk or wherever you program.

[N+07] "Exterminator: Automatically Correcting Memory Errors with High Probability"
Gene Novark, Emery D. Berger, and Benjamin G. Zorn
PLDI 2007
A cool paper on finding and correcting memory errors automatically, and a great overview of many common errors in C and C++ programs.

[SN05] "Using Valgrind to Detect Undefined Value Errors with Bit-precision"
J. Seward and N. Nethercote
USENIX '05
How to use valgrind to find certain types of errors.

[SR05] "Advanced Programming in the UNIX Environment"
W. Richard Stevens and Stephen A. Rago
Addison-Wesley, 2005
We've said it before, we'll say it again: read this book many times and use it as a reference whenever you are in doubt. The authors are always surprised at how each time they read something in this book, they learn something new, even after many years of C programming.

[W06] "Survey on Buffer Overflow Attacks and Countermeasures"
Tim Werthman
Available: www.nds.rub.de/lehre/seminar/SS06/Werthmann_BufferOverflow.pdf
A nice survey of buffer overflows and some of the security problems they cause. Refers to many of the famous exploits.

Homework (Code)

In this homework, you will gain some familiarity with memory allocation. First, you'll write some buggy programs (fun!). Then, you'll use some tools to help you find the bugs you inserted. Then, you will realize how awesome these tools are and use them in the future, thus making yourself more happy and productive.

The first tool you'll use is `gdb`, the debugger. There is a lot to learn about this debugger; here we'll only scratch the surface.

The second tool you'll use is `valgrind` [SN05]. This tool helps find memory leaks and other insidious memory problems in your program. If it's not installed on your system, go to the website and do so:

```
http://valgrind.org/downloads/current.html
```

Questions

1. First, write a simple program called `null.c` that creates a pointer to an integer, sets it to `NULL`, and then tries to dereference it. Compile this into an executable called `null`. What happens when you run this program?
2. Next, compile this program with symbol information included (with the `-g` flag). Doing so let's put more information into the executable, enabling the debugger to access more useful information about variable names and the like. Run the program under the debugger by typing `gdb null` and then, once `gdb` is running, typing `run`. What does gdb show you?
3. Finally, use the `valgrind` tool on this program. We'll use the `memcheck` tool that is a part of `valgrind` to analyze what happens. Run this by typing in the following: `valgrind --leak-check=yes null`. What happens when you run this? Can you interpret the output from the tool?
4. Write a simple program that allocates memory using `malloc()` but forgets to free it before exiting. What happens when this program runs? Can you use `gdb` to find any problems with it? How about `valgrind` (again with the `--leak-check=yes` flag)?
5. Write a program that creates an array of integers called `data` of size 100 using `malloc`; then, set `data[100]` to zero. What happens when you run this program? What happens when you run this program using `valgrind`? Is the program correct?
6. Create a program that allocates an array of integers (as above), frees them, and then tries to print the value of one of the elements of the array. Does the program run? What happens when you use `valgrind` on it?
7. Now pass a funny value to free (e.g., a pointer in the middle of the array you allocated above). What happens? Do you need tools to find this type of problem?

8. Try out some of the other interfaces to memory allocation. For example, create a simple vector-like data structure and related routines that use `realloc()` to manage the vector. Use an array to store the vectors elements; when a user adds an entry to the vector, use `realloc()` to allocate more space for it. How well does such a vector perform? How does it compare to a linked list? Use `valgrind` to help you find bugs.
9. Spend more time and read about using `gdb` and `valgrind`. Knowing your tools is critical; spend the time and learn how to become an expert debugger in the UNIX and C environment.

15

Mechanism: Address Translation

In developing the virtualization of the CPU, we focused on a general mechanism known as **limited direct execution** (or **LDE**). The idea behind LDE is simple: for the most part, let the program run directly on the hardware; however, at certain key points in time (such as when a process issues a system call, or a timer interrupt occurs), arrange so that the OS gets involved and makes sure the "right" thing happens. Thus, the OS, with a little hardware support, tries its best to get out of the way of the running program, to deliver an *efficient* virtualization; however, by **interposing** at those critical points in time, the OS ensures that it maintains *control* over the hardware. Efficiency and control together are two of the main goals of any modern operating system.

In virtualizing memory, we will pursue a similar strategy, attaining both efficiency and control while providing the desired virtualization. Efficiency dictates that we make use of hardware support, which at first will be quite rudimentary (e.g., just a few registers) but will grow to be fairly complex (e.g., TLBs, page-table support, and so forth, as you will see). Control implies that the OS ensures that no application is allowed to access any memory but its own; thus, to protect applications from one another, and the OS from applications, we will need help from the hardware here too. Finally, we will need a little more from the VM system, in terms of *flexibility*; specifically, we'd like for programs to be able to use their address spaces in whatever way they would like, thus making the system easier to program. And thus we arrive at the refined crux:

THE CRUX:
HOW TO EFFICIENTLY AND FLEXIBLY VIRTUALIZE MEMORY

How can we build an efficient virtualization of memory? How do we provide the flexibility needed by applications? How do we maintain control over which memory locations an application can access, and thus ensure that application memory accesses are properly restricted? How do we do all of this efficiently?

The generic technique we will use, which you can consider an addition to our general approach of limited direct execution, is something that is referred to as **hardware-based address translation**, or just **address translation** for short. With address translation, the hardware transforms each memory access (e.g., an instruction fetch, load, or store), changing the **virtual** address provided by the instruction to a **physical** address where the desired information is actually located. Thus, on each and every memory reference, an address translation is performed by the hardware to redirect application memory references to their actual locations in memory.

Of course, the hardware alone cannot virtualize memory, as it just provides the low-level mechanism for doing so efficiently. The OS must get involved at key points to set up the hardware so that the correct translations take place; it must thus **manage memory**, keeping track of which locations are free and which are in use, and judiciously intervening to maintain control over how memory is used.

Once again the goal of all of this work is to create a beautiful **illusion**: that the program has its own private memory, where its own code and data reside. Behind that virtual reality lies the ugly physical truth: that many programs are actually sharing memory at the same time, as the CPU (or CPUs) switches between running one program and the next. Through virtualization, the OS (with the hardware's help) turns the ugly machine reality into something that is a useful, powerful, and easy to use abstraction.

15.1 Assumptions

Our first attempts at virtualizing memory will be very simple, almost laughably so. Go ahead, laugh all you want; pretty soon it will be the OS laughing at you, when you try to understand the ins and outs of TLBs, multi-level page tables, and other technical wonders. Don't like the idea of the OS laughing at you? Well, you may be out of luck then; that's just how the OS rolls.

Specifically, we will assume for now that the user's address space must be placed *contiguously* in physical memory. We will also assume, for simplicity, that the size of the address space is not too big; specifically, that it is *less than the size of physical memory*. Finally, we will also assume that each address space is exactly the *same size*. Don't worry if these assumptions sound unrealistic; we will relax them as we go, thus achieving a realistic virtualization of memory.

15.2 An Example

To understand better what we need to do to implement address translation, and why we need such a mechanism, let's look at a simple example. Imagine there is a process whose address space is as indicated in Figure 15.1. What we are going to examine here is a short code sequence

TIP: INTERPOSITION IS POWERFUL

Interposition is a generic and powerful technique that is often used to great effect in computer systems. In virtualizing memory, the hardware will interpose on each memory access, and translate each virtual address issued by the process to a physical address where the desired information is actually stored. However, the general technique of interposition is much more broadly applicable; indeed, almost any well-defined interface can be interposed upon, to add new functionality or improve some other aspect of the system. One of the usual benefits of such an approach is **transparency**; the interposition often is done without changing the client of the interface, thus requiring no changes to said client.

that loads a value from memory, increments it by three, and then stores the value back into memory. You can imagine the C-language representation of this code might look like this:

```
void func() {
    int x = 3000; // thanks, Perry.
    x = x + 3;    // this is the line of code we are interested in
```

The compiler turns this line of code into assembly, which might look something like this (in x86 assembly). Use `objdump` on Linux or `otool` on Mac OS X to disassemble it:

```
128: movl 0x0(%ebx), %eax     ;load 0+ebx into eax
132: addl $0x03, %eax         ;add 3 to eax register
135: movl %eax, 0x0(%ebx)     ;store eax back to mem
```

This code snippet is relatively straightforward; it presumes that the address of `x` has been placed in the register `ebx`, and then loads the value at that address into the general-purpose register `eax` using the `movl` instruction (for "longword" move). The next instruction adds 3 to `eax`, and the final instruction stores the value in `eax` back into memory at that same location.

In Figure 15.1 (page 4), you can see how both the code and data are laid out in the process's address space; the three-instruction code sequence is located at address 128 (in the code section near the top), and the value of the variable `x` at address 15 KB (in the stack near the bottom). In the figure, the initial value of `x` is 3000, as shown in its location on the stack.

When these instructions run, from the perspective of the process, the following memory accesses take place.

- Fetch instruction at address 128
- Execute this instruction (load from address 15 KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

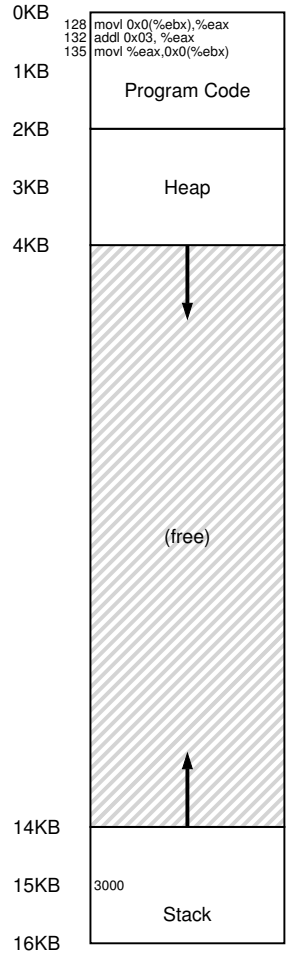


Figure 15.1: **A Process And Its Address Space**

From the program's perspective, its **address space** starts at address 0 and grows to a maximum of 16 KB; all memory references it generates should be within these bounds. However, to virtualize memory, the OS wants to place the process somewhere else in physical memory, not necessarily at address 0. Thus, we have the problem: how can we **relocate** this process in memory in a way that is **transparent** to the process? How can we provide the illusion of a virtual address space starting at 0, when in reality the address space is located at some other physical address?

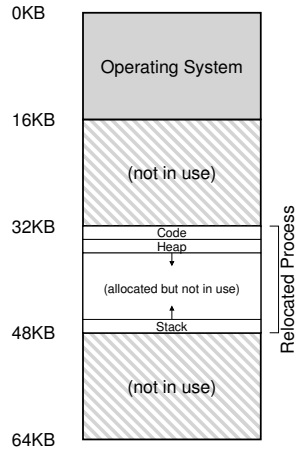


Figure 15.2: **Physical Memory with a Single Relocated Process**

An example of what physical memory might look like once this process's address space has been placed in memory is found in Figure 15.2. In the figure, you can see the OS using the first slot of physical memory for itself, and that it has relocated the process from the example above into the slot starting at physical memory address 32 KB. The other two slots are free (16 KB-32 KB and 48 KB-64 KB).

15.3 Dynamic (Hardware-based) Relocation

To gain some understanding of hardware-based address translation, we'll first discuss its first incarnation. Introduced in the first time-sharing machines of the late 1950's is a simple idea referred to as **base and bounds**; the technique is also referred to as **dynamic relocation**; we'll use both terms interchangeably [SS74].

Specifically, we'll need two hardware registers within each CPU: one is called the **base** register, and the other the **bounds** (sometimes called a **limit** register). This base-and-bounds pair is going to allow us to place the address space anywhere we'd like in physical memory, and do so while ensuring that the process can only access its own address space.

In this setup, each program is written and compiled as if it is loaded at address zero. However, when a program starts running, the OS decides where in physical memory it should be loaded and sets the base register to that value. In the example above, the OS decides to load the process at physical address 32 KB and thus sets the base register to this value.

Interesting things start to happen when the process is running. Now, when any memory reference is generated by the process, it is **translated** by the processor in the following manner:

```
physical address = virtual address + base
```

ASIDE: **SOFTWARE-BASED RELOCATION**

In the early days, before hardware support arose, some systems performed a crude form of relocation purely via software methods. The basic technique is referred to as **static relocation**, in which a piece of software known as the **loader** takes an executable that is about to be run and rewrites its addresses to the desired offset in physical memory.

For example, if an instruction was a load from address 1000 into a register (e.g., `movl 1000, %eax`), and the address space of the program was loaded starting at address 3000 (and not 0, as the program thinks), the loader would rewrite the instruction to offset each address by 3000 (e.g., `movl 4000, %eax`). In this way, a simple static relocation of the process's address space is achieved.

However, static relocation has numerous problems. First and most importantly, it does not provide protection, as processes can generate bad addresses and thus illegally access other process's or even OS memory; in general, hardware support is likely needed for true protection [WL+93]. Another negative is that once placed, it is difficult to later relocate an address space to another location [M65].

Each memory reference generated by the process is a **virtual address**; the hardware in turn adds the contents of the base register to this address and the result is a **physical address** that can be issued to the memory system.

To understand this better, let's trace through what happens when a single instruction is executed. Specifically, let's look at one instruction from our earlier sequence:

```
128: movl 0x0(%ebx), %eax
```

The program counter (PC) is set to 128; when the hardware needs to fetch this instruction, it first adds the value to the base register value of 32 KB (32768) to get a physical address of 32896; the hardware then fetches the instruction from that physical address. Next, the processor begins executing the instruction. At some point, the process then issues the load from virtual address 15 KB, which the processor takes and again adds to the base register (32 KB), getting the final physical address of 47 KB and thus the desired contents.

Transforming a virtual address into a physical address is exactly the technique we refer to as **address translation**; that is, the hardware takes a virtual address the process thinks it is referencing and transforms it into a physical address which is where the data actually resides. Because this relocation of the address happens at runtime, and because we can move address spaces even after the process has started running, the technique is often referred to as **dynamic relocation** [M65].

TIP: HARDWARE-BASED DYNAMIC RELOCATION
With dynamic relocation, a little hardware goes a long way. Namely, a **base** register is used to transform virtual addresses (generated by the program) into physical addresses. A **bounds** (or **limit**) register ensures that such addresses are within the confines of the address space. Together they provide a simple and efficient virtualization of memory.

Now you might be asking: what happened to that bounds (limit) register? After all, isn't this the base *and* bounds approach? Indeed, it is. As you might have guessed, the bounds register is there to help with protection. Specifically, the processor will first check that the memory reference is *within bounds* to make sure it is legal; in the simple example above, the bounds register would always be set to 16 KB. If a process generates a virtual address that is greater than the bounds, or one that is negative, the CPU will raise an exception, and the process will likely be terminated. The point of the bounds is thus to make sure that all addresses generated by the process are legal and within the "bounds" of the process.

We should note that the base and bounds registers are hardware structures kept on the chip (one pair per CPU). Sometimes people call the part of the processor that helps with address translation the **memory management unit (MMU)**; as we develop more sophisticated memory-management techniques, we will be adding more circuitry to the MMU.

A small aside about bound registers, which can be defined in one of two ways. In one way (as above), it holds the *size* of the address space, and thus the hardware checks the virtual address against it first before adding the base. In the second way, it holds the *physical address* of the end of the address space, and thus the hardware first adds the base and then makes sure the address is within bounds. Both methods are logically equivalent; for simplicity, we'll usually assume the former method.

Example Translations

To understand address translation via base-and-bounds in more detail, let's take a look at an example. Imagine a process with an address space of size 4 KB (yes, unrealistically small) has been loaded at physical address 16 KB. Here are the results of a number of address translations:

Virtual Address		Physical Address
0	→	16 KB
1 KB	→	17 KB
3000	→	19384
4400	→	*Fault (out of bounds)*

As you can see from the example, it is easy for you to simply add the base address to the virtual address (which can rightly be viewed as an *offset* into the address space) to get the resulting physical address. Only if the virtual address is "too big" or negative will the result be a fault, causing an exception to be raised.

ASIDE: **DATA STRUCTURE — THE FREE LIST**
The OS must track which parts of free memory are not in use, so as to be able to allocate memory to processes. Many different data structures can of course be used for such a task; the simplest (which we will assume here) is a **free list**, which simply is a list of the ranges of the physical memory which are not currently in use.

15.4 Hardware Support: A Summary

Let us now summarize the support we need from the hardware (also see Figure 15.3, page 9). First, as discussed in the chapter on CPU virtualization, we require two different CPU modes. The OS runs in **privileged mode** (or **kernel mode**), where it has access to the entire machine; applications run in **user mode**, where they are limited in what they can do. A single bit, perhaps stored in some kind of **processor status word**, indicates which mode the CPU is currently running in; upon certain special occasions (e.g., a system call or some other kind of exception or interrupt), the CPU switches modes.

The hardware must also provide the **base and bounds registers** themselves; each CPU thus has an additional pair of registers, part of the **memory management unit** (**MMU**) of the CPU. When a user program is running, the hardware will translate each address, by adding the base value to the virtual address generated by the user program. The hardware must also be able to check whether the address is valid, which is accomplished by using the bounds register and some circuitry within the CPU.

The hardware should provide special instructions to modify the base and bounds registers, allowing the OS to change them when different processes run. These instructions are **privileged**; only in kernel (or privileged) mode can the registers be modified. Imagine the havoc a user process could wreak[1] if it could arbitrarily change the base register while running. Imagine it! And then quickly flush such dark thoughts from your mind, as they are the ghastly stuff of which nightmares are made.

Finally, the CPU must be able to generate **exceptions** in situations where a user program tries to access memory illegally (with an address that is "out of bounds"); in this case, the CPU should stop executing the user program and arrange for the OS "out-of-bounds" **exception handler** to run. The OS handler can then figure out how to react, in this case likely terminating the process. Similarly, if a user program tries to change the values of the (privileged) base and bounds registers, the CPU should raise an exception and run the "tried to execute a privileged operation while in user mode" handler. The CPU also must provide a method to inform it of the location of these handlers; a few more privileged instructions are thus needed.

[1] Is there anything other than "havoc" that can be "wreaked"?

Hardware Requirements	Notes
Privileged mode	*Needed to prevent user-mode processes from executing privileged operations*
Base/bounds registers	*Need pair of registers per CPU to support address translation and bounds checks*
Ability to translate virtual addresses and check if within bounds	*Circuitry to do translations and check limits; in this case, quite simple*
Privileged instruction(s) to update base/bounds	*OS must be able to set these values before letting a user program run*
Privileged instruction(s) to register exception handlers	*OS must be able to tell hardware what code to run if exception occurs*
Ability to raise exceptions	*When processes try to access privileged instructions or out-of-bounds memory*

Figure 15.3: **Dynamic Relocation: Hardware Requirements**

15.5 Operating System Issues

Just as the hardware provides new features to support dynamic relocation, the OS now has new issues it must handle; the combination of hardware support and OS management leads to the implementation of a simple virtual memory. Specifically, there are a few critical junctures where the OS must get involved to implement our base-and-bounds version of virtual memory.

First, the OS must take action when a process is created, finding space for its address space in memory. Fortunately, given our assumptions that each address space is (a) smaller than the size of physical memory and (b) the same size, this is quite easy for the OS; it can simply view physical memory as an array of slots, and track whether each one is free or in use. When a new process is created, the OS will have to search a data structure (often called a **free list**) to find room for the new address space and then mark it used. With variable-sized address spaces, life is more complicated, but we will leave that concern for future chapters.

Let's look at an example. In Figure 15.2 (page 5), you can see the OS using the first slot of physical memory for itself, and that it has relocated the process from the example above into the slot starting at physical memory address 32 KB. The other two slots are free (16 KB-32 KB and 48 KB-64 KB); thus, the **free list** should consist of these two entries.

Second, the OS must do some work when a process is terminated (i.e., when it exits gracefully, or is forcefully killed because it misbehaved), reclaiming all of its memory for use in other processes or the OS. Upon termination of a process, the OS thus puts its memory back on the free list, and cleans up any associated data structures as need be.

Third, the OS must also perform a few additional steps when a context switch occurs. There is only one base and bounds register pair on each CPU, after all, and their values differ for each running program, as each program is loaded at a different physical address in memory. Thus, the OS must *save and restore* the base-and-bounds pair when it switches be-

OS Requirements	Notes
Memory management	*Need to allocate memory for new processes;* *Reclaim memory from terminated processes;* *Generally manage memory via **free list***
Base/bounds management	*Must set base/bounds properly upon context switch*
Exception handling	*Code to run when exceptions arise;* *likely action is to terminate offending process*

Figure 15.4: **Dynamic Relocation: Operating System Responsibilities**

tween processes. Specifically, when the OS decides to stop running a process, it must save the values of the base and bounds registers to memory, in some per-process structure such as the **process structure** or **process control block** (PCB). Similarly, when the OS resumes a running process (or runs it the first time), it must set the values of the base and bounds on the CPU to the correct values for this process.

We should note that when a process is stopped (i.e., not running), it is possible for the OS to move an address space from one location in memory to another rather easily. To move a process's address space, the OS first deschedules the process; then, the OS copies the address space from the current location to the new location; finally, the OS updates the saved base register (in the process structure) to point to the new location. When the process is resumed, its (new) base register is restored, and it begins running again, oblivious that its instructions and data are now in a completely new spot in memory.

Fourth, the OS must provide **exception handlers**, or functions to be called, as discussed above; the OS installs these handlers at boot time (via privileged instructions). For example, if a process tries to access memory outside its bounds, the CPU will raise an exception; the OS must be prepared to take action when such an exception arises. The common reaction of the OS will be one of hostility: it will likely terminate the offending process. The OS should be highly protective of the machine it is running, and thus it does not take kindly to a process trying to access memory or execute instructions that it shouldn't. Bye bye, misbehaving process; it's been nice knowing you.

Figure 15.5 (page 11) illustrates much of the hardware/OS interaction in a timeline. The figure shows what the OS does at boot time to ready the machine for use, and then what happens when a process (Process A) starts running; note how its memory translations are handled by the hardware with no OS intervention. At some point, a timer interrupt occurs, and the OS switches to Process B, which executes a "bad load" (to an illegal memory address); at that point, the OS must get involved, terminating the process and cleaning up by freeing B's memory and removing its entry from the process table. As you can see from the diagram, we are still following the basic approach of **limited direct execution**. In most cases, the OS just sets up the hardware appropriately and lets the process run directly on the CPU; only when the process misbehaves does the OS have to become involved.

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember addresses of...	
	system call handler	
	timer handler	
	illegal mem-access handler	
	illegal instruction handler	
start interrupt timer		
	start timer; interrupt after X ms	
initialize process table		
initialize free list		

OS @ run (kernel mode)	Hardware	Program (user mode)
To start process A:		
allocate entry in process table		
allocate memory for process		
set base/bounds registers		
return-from-trap (into A)		
	restore registers of A	
	move to **user mode**	
	jump to A's (initial) PC	
		Process A runs
		Fetch instruction
	Translate virtual address	
	and perform fetch	
		Execute instruction
	If explicit load/store:	
	Ensure address is in-bounds;	
	Translate virtual address	
	and perform load/store	
		...
	Timer interrupt	
	move to **kernel mode**	
	Jump to interrupt handler	
Handle the trap		
Call `switch()` routine		
save regs(A) to proc-struct(A)		
(including base/bounds)		
restore regs(B) from proc-struct(B)		
(including base/bounds)		
return-from-trap (into B)		
	restore registers of B	
	move to **user mode**	
	jump to B's PC	
		Process B runs
		Execute bad load
	Load is out-of-bounds;	
	move to **kernel mode**	
	jump to trap handler	
Handle the trap		
Decide to terminate process B		
de-allocate B's memory		
free B's entry in process table		

Figure 15.5: **Limited Direct Execution Protocol (Dynamic Relocation)**

15.6 Summary

In this chapter, we have extended the concept of limited direct execution with a specific mechanism used in virtual memory, known as **address translation**. With address translation, the OS can control each and every memory access from a process, ensuring the accesses stay within the bounds of the address space. Key to the efficiency of this technique is hardware support, which performs the translation quickly for each access, turning virtual addresses (the process's view of memory) into physical ones (the actual view). All of this is performed in a way that is *transparent* to the process that has been relocated; the process has no idea its memory references are being translated, making for a wonderful illusion.

We have also seen one particular form of virtualization, known as base and bounds or dynamic relocation. Base-and-bounds virtualization is quite *efficient*, as only a little more hardware logic is required to add a base register to the virtual address and check that the address generated by the process is in bounds. Base-and-bounds also offers *protection*; the OS and hardware combine to ensure no process can generate memory references outside its own address space. Protection is certainly one of the most important goals of the OS; without it, the OS could not control the machine (if processes were free to overwrite memory, they could easily do nasty things like overwrite the trap table and take over the system).

Unfortunately, this simple technique of dynamic relocation does have its inefficiencies. For example, as you can see in Figure 15.2 (page 5), the relocated process is using physical memory from 32 KB to 48 KB; however, because the process stack and heap are not too big, all of the space between the two is simply *wasted*. This type of waste is usually called **internal fragmentation**, as the space *inside* the allocated unit is not all used (i.e., is fragmented) and thus wasted. In our current approach, although there might be enough physical memory for more processes, we are currently restricted to placing an address space in a fixed-sized slot and thus internal fragmentation can arise[2]. Thus, we are going to need more sophisticated machinery, to try to better utilize physical memory and avoid internal fragmentation. Our first attempt will be a slight generalization of base and bounds known as **segmentation**, which we will discuss next.

[2] A different solution might instead place a fixed-sized stack within the address space, just below the code region, and a growing heap below that. However, this limits flexibility by making recursion and deeply-nested function calls challenging, and thus is something we hope to avoid.

References

[M65] "On Dynamic Program Relocation"
W.C. McGee
IBM Systems Journal
Volume 4, Number 3, 1965, pages 184–199
This paper is a nice summary of early work on dynamic relocation, as well as some basics on static relocation.

[P90] "Relocating loader for MS-DOS .EXE executable files"
Kenneth D. A. Pillay
Microprocessors & Microsystems archive
Volume 14, Issue 7 (September 1990)
An example of a relocating loader for MS-DOS. Not the first one, but just a relatively modern example of how such a system works.

[SS74] "The Protection of Information in Computer Systems"
J. Saltzer and M. Schroeder
CACM, July 1974
From this paper: "The concepts of base-and-bound register and hardware-interpreted descriptors appeared, apparently independently, between 1957 and 1959 on three projects with diverse goals. At M.I.T., McCarthy suggested the base-and-bound idea as part of the memory protection system necessary to make time-sharing feasible. IBM independently developed the base-and-bound register as a mechanism to permit reliable multiprogramming of the Stretch (7030) computer system. At Burroughs, R. Barton suggested that hardware-interpreted descriptors would provide direct support for the naming scope rules of higher level languages in the B5000 computer system." We found this quote on Mark Smotherman's cool history pages [S04]; see them for more information.

[S04] "System Call Support"
Mark Smotherman, May 2004
http://people.cs.clemson.edu/~mark/syscall.html
A neat history of system call support. Smotherman has also collected some early history on items like interrupts and other fun aspects of computing history. See his web pages for more details.

[WL+93] "Efficient Software-based Fault Isolation"
Robert Wahbe, Steven Lucco, Thomas E. Anderson, Susan L. Graham
SOSP '93
A terrific paper about how you can use compiler support to bound memory references from a program, without hardware support. The paper sparked renewed interest in software techniques for isolation of memory references.

Homework

The program `relocation.py` allows you to see how address translations are performed in a system with base and bounds registers. See the README for details.

Questions

1. Run with seeds 1, 2, and 3, and compute whether each virtual address generated by the process is in or out of bounds. If in bounds, compute the translation.
2. Run with these flags: `-s 0 -n 10`. What value do you have set `-l` (the bounds register) to in order to ensure that all the generated virtual addresses are within bounds?
3. Run with these flags: `-s 1 -n 10 -l 100`. What is the maximum value that base can be set to, such that the address space still fits into physical memory in its entirety?
4. Run some of the same problems above, but with larger address spaces (`-a`) and physical memories (`-p`).
5. What fraction of randomly-generated virtual addresses are valid, as a function of the value of the bounds register? Make a graph from running with different random seeds, with limit values ranging from 0 up to the maximum size of the address space.

16

Segmentation

So far we have been putting the entire address space of each process in memory. With the base and bounds registers, the OS can easily relocate processes to different parts of physical memory. However, you might have noticed something interesting about these address spaces of ours: there is a big chunk of "free" space right in the middle, between the stack and the heap.

As you can imagine from Figure 16.1, although the space between the stack and heap is not being used by the process, it is still taking up physical memory when we relocate the entire address space somewhere in physical memory; thus, the simple approach of using a base and bounds register pair to virtualize memory is wasteful. It also makes it quite hard to run a program when the entire address space doesn't fit into memory; thus, base and bounds is not as flexible as we would like. And thus:

THE CRUX: HOW TO SUPPORT A LARGE ADDRESS SPACE

How do we support a large address space with (potentially) a lot of free space between the stack and the heap? Note that in our examples, with tiny (pretend) address spaces, the waste doesn't seem too bad. Imagine, however, a 32-bit address space (4 GB in size); a typical program will only use megabytes of memory, but still would demand that the entire address space be resident in memory.

16.1 Segmentation: Generalized Base/Bounds

To solve this problem, an idea was born, and it is called **segmentation**. It is quite an old idea, going at least as far back as the very early 1960's [H61, G62]. The idea is simple: instead of having just one base and bounds pair in our MMU, why not have a base and bounds pair per logical **segment** of the address space? A segment is just a contiguous portion of the address space of a particular length, and in our canonical

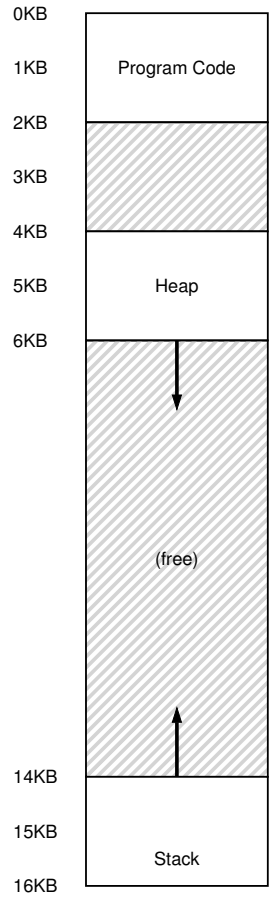


Figure 16.1: **An Address Space (Again)**

address space, we have three logically-different segments: code, stack, and heap. What segmentation allows the OS to do is to place each one of those segments in different parts of physical memory, and thus avoid filling physical memory with unused virtual address space.

Let's look at an example. Assume we want to place the address space from Figure 16.1 into physical memory. With a base and bounds pair per segment, we can place each segment *independently* in physical memory. For example, see Figure 16.2 (page 3); there you see a 64KB physical memory with those three segments in it (and 16KB reserved for the OS).

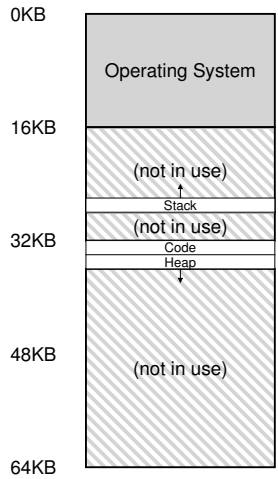


Figure 16.2: **Placing Segments In Physical Memory**

As you can see in the diagram, only used memory is allocated space in physical memory, and thus large address spaces with large amounts of unused address space (which we sometimes call **sparse address spaces**) can be accommodated.

The hardware structure in our MMU required to support segmentation is just what you'd expect: in this case, a set of three base and bounds register pairs. Figure 16.3 below shows the register values for the example above; each bounds register holds the size of a segment.

Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

Figure 16.3: **Segment Register Values**

You can see from the figure that the code segment is placed at physical address 32KB and has a size of 2KB and the heap segment is placed at 34KB and also has a size of 2KB.

Let's do an example translation, using the address space in Figure 16.1. Assume a reference is made to virtual address 100 (which is in the code segment). When the reference takes place (say, on an instruction fetch), the hardware will add the base value to the *offset* into this segment (100 in this case) to arrive at the desired physical address: 100 + 32KB, or 32868. It will then check that the address is within bounds (100 is less than 2KB), find that it is, and issue the reference to physical memory address 32868.

ASIDE: **THE SEGMENTATION FAULT**
The term segmentation fault or violation arises from a memory access on a segmented machine to an illegal address. Humorously, the term persists, even on machines with no support for segmentation at all. Or not so humorously, if you can't figure why your code keeps faulting.

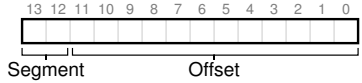
Now let's look at an address in the heap, virtual address 4200 (again refer to Figure 16.1). If we just add the virtual address 4200 to the base of the heap (34KB), we get a physical address of 39016, which is *not* the correct physical address. What we need to first do is extract the *offset* into the heap, i.e., which byte(s) *in this segment* the address refers to. Because the heap starts at virtual address 4KB (4096), the offset of 4200 is actually 4200 minus 4096, or 104. We then take this offset (104) and add it to the base register physical address (34K) to get the desired result: 34920.

What if we tried to refer to an illegal address, such as 7KB which is beyond the end of the heap? You can imagine what will happen: the hardware detects that the address is out of bounds, traps into the OS, likely leading to the termination of the offending process. And now you know the origin of the famous term that all C programmers learn to dread: the **segmentation violation** or **segmentation fault**.

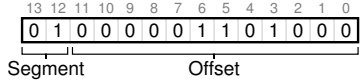
16.2 Which Segment Are We Referring To?

The hardware uses segment registers during translation. How does it know the offset into a segment, and to which segment an address refers?

One common approach, sometimes referred to as an **explicit** approach, is to chop up the address space into segments based on the top few bits of the virtual address; this technique was used in the VAX/VMS system [LL82]. In our example above, we have three segments; thus we need two bits to accomplish our task. If we use the top two bits of our 14-bit virtual address to select the segment, our virtual address looks like this:



In our example, then, if the top two bits are 00, the hardware knows the virtual address is in the code segment, and thus uses the code base and bounds pair to relocate the address to the correct physical location. If the top two bits are 01, the hardware knows the address is in the heap, and thus uses the heap base and bounds. Let's take our example heap virtual address from above (4200) and translate it, just to make sure this is clear. The virtual address 4200, in binary form, can be seen here:



As you can see from the picture, the top two bits (01) tell the hardware which *segment* we are referring to. The bottom 12 bits are the *offset* into the segment: 0000 0110 1000, or hex 0x068, or 104 in decimal. Thus, the hardware simply takes the first two bits to determine which segment register to use, and then takes the next 12 bits as the offset into the segment. By adding the base register to the offset, the hardware arrives at the final physical address. Note the offset eases the bounds check too: we can simply check if the offset is less than the bounds; if not, the address is illegal. Thus, if base and bounds were arrays (with one entry per segment), the hardware would be doing something like this to obtain the desired physical address:

```
1 // get top 2 bits of 14-bit VA
2 Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3 // now get offset
4 Offset  = VirtualAddress & OFFSET_MASK
5 if (Offset >= Bounds[Segment])
6     RaiseException(PROTECTION_FAULT)
7 else
8     PhysAddr = Base[Segment] + Offset
9     Register = AccessMemory(PhysAddr)
```

In our running example, we can fill in values for the constants above. Specifically, `SEG_MASK` would be set to `0x3000`, `SEG_SHIFT` to `12`, and `OFFSET_MASK` to `0xFFF`.

You may also have noticed that when we use the top two bits, and we only have three segments (code, heap, stack), one segment of the address space goes unused. Thus, some systems put code in the same segment as the heap and thus use only one bit to select which segment to use [LL82].

There are other ways for the hardware to determine which segment a particular address is in. In the **implicit** approach, the hardware determines the segment by noticing how the address was formed. If, for example, the address was generated from the program counter (i.e., it was an instruction fetch), then the address is within the code segment; if the address is based off of the stack or base pointer, it must be in the stack segment; any other address must be in the heap.

16.3 What About The Stack?

Thus far, we've left out one important component of the address space: the stack. The stack has been relocated to physical address 28KB in the diagram above, but with one critical difference: *it grows backwards*. In physical memory, it starts at 28KB and grows back to 26KB, corresponding to virtual addresses 16KB to 14KB; translation must proceed differently.

The first thing we need is a little extra hardware support. Instead of just base and bounds values, the hardware also needs to know which way the segment grows (a bit, for example, that is set to 1 when the segment grows in the positive direction, and 0 for negative). Our updated view of what the hardware tracks is seen in Figure 16.4.

Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0

Figure 16.4: **Segment Registers (With Negative-Growth Support)**

With the hardware understanding that segments can grow in the negative direction, the hardware must now translate such virtual addresses slightly differently. Let's take an example stack virtual address and translate it to understand the process.

In this example, assume we wish to access virtual address 15KB, which should map to physical address 27KB. Our virtual address, in binary form, thus looks like this: 11 1100 0000 0000 (hex 0x3C00). The hardware uses the top two bits (11) to designate the segment, but then we are left with an offset of 3KB. To obtain the correct negative offset, we must subtract the maximum segment size from 3KB: in this example, a segment can be 4KB, and thus the correct negative offset is 3KB minus 4KB which equals -1KB. We simply add the negative offset (-1KB) to the base (28KB) to arrive at the correct physical address: 27KB. The bounds check can be calculated by ensuring the absolute value of the negative offset is less than the segment's size.

16.4 Support for Sharing

As support for segmentation grew, system designers soon realized that they could realize new types of efficiencies with a little more hardware support. Specifically, to save memory, sometimes it is useful to **share** certain memory segments between address spaces. In particular, **code sharing** is common and still in use in systems today.

To support sharing, we need a little extra support from the hardware, in the form of **protection bits**. Basic support adds a few bits per segment, indicating whether or not a program can read or write a segment, or perhaps execute code that lies within the segment. By setting a code segment to read-only, the same code can be shared across multiple processes, without worry of harming isolation; while each process still thinks that it is accessing its own private memory, the OS is secretly sharing memory which cannot be modified by the process, and thus the illusion is preserved.

An example of the additional information tracked by the hardware (and OS) is shown in Figure 16.5. As you can see, the code segment is set to read and execute, and thus the same physical segment in memory could be mapped into multiple virtual address spaces.

Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

Figure 16.5: **Segment Register Values (with Protection)**

With protection bits, the hardware algorithm described earlier would also have to change. In addition to checking whether a virtual address is within bounds, the hardware also has to check whether a particular access is permissible. If a user process tries to write to a read-only segment, or execute from a non-executable segment, the hardware should raise an exception, and thus let the OS deal with the offending process.

16.5 Fine-grained vs. Coarse-grained Segmentation

Most of our examples thus far have focused on systems with just a few segments (i.e., code, stack, heap); we can think of this segmentation as **coarse-grained**, as it chops up the address space into relatively large, coarse chunks. However, some early systems (e.g., Multics [CV65,DD68]) were more flexible and allowed for address spaces to consist of a large number of smaller segments, referred to as **fine-grained** segmentation.

Supporting many segments requires even further hardware support, with a **segment table** of some kind stored in memory. Such segment tables usually support the creation of a very large number of segments, and thus enable a system to use segments in more flexible ways than we have thus far discussed. For example, early machines like the Burroughs B5000 had support for thousands of segments, and expected a compiler to chop code and data into separate segments which the OS and hardware would then support [RK68]. The thinking at the time was that by having fine-grained segments, the OS could better learn about which segments are in use and which are not and thus utilize main memory more effectively.

16.6 OS Support

You now should have a basic idea as to how segmentation works. Pieces of the address space are relocated into physical memory as the system runs, and thus a huge savings of physical memory is achieved relative to our simpler approach with just a single base/bounds pair for the entire address space. Specifically, all the unused space between the stack and the heap need not be allocated in physical memory, allowing us to fit more address spaces into physical memory.

However, segmentation raises a number of new issues. We'll first describe the new OS issues that must be addressed. The first is an old one: what should the OS do on a context switch? You should have a good guess by now: the segment registers must be saved and restored. Clearly, each process has its own virtual address space, and the OS must make sure to set up these registers correctly before letting the process run again.

The second, and more important, issue is managing free space in physical memory. When a new address space is created, the OS has to be able to find space in physical memory for its segments. Previously, we assumed that each address space was the same size, and thus physical memory could be thought of as a bunch of slots where processes would fit in. Now, we have a number of segments per process, and each segment might be a different size.

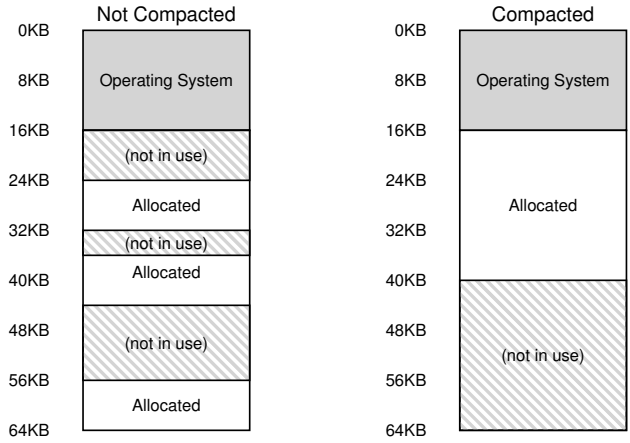


Figure 16.6: **Non-compacted and Compacted Memory**

The general problem that arises is that physical memory quickly becomes full of little holes of free space, making it difficult to allocate new segments, or to grow existing ones. We call this problem **external fragmentation** [R69]; see Figure 16.6 (left).

In the example, a process comes along and wishes to allocate a 20KB segment. In that example, there is 24KB free, but not in one contiguous segment (rather, in three non-contiguous chunks). Thus, the OS cannot satisfy the 20KB request.

One solution to this problem would be to **compact** physical memory by rearranging the existing segments. For example, the OS could stop whichever processes are running, copy their data to one contiguous region of memory, change their segment register values to point to the new physical locations, and thus have a large free extent of memory with which to work. By doing so, the OS enables the new allocation request to succeed. However, compaction is expensive, as copying segments is memory-intensive and generally uses a fair amount of processor time. See Figure 16.6 (right) for a diagram of compacted physical memory.

A simpler approach is to use a free-list management algorithm that tries to keep large extents of memory available for allocation. There are literally hundreds of approaches that people have taken, including classic algorithms like **best-fit** (which keeps a list of free spaces and returns the one closest in size that satisfies the desired allocation to the requester), **worst-fit**, **first-fit**, and more complex schemes like **buddy algorithm** [K68]. An excellent survey by Wilson et al. is a good place to start if you want to learn more about such algorithms [W+95], or you can wait until we cover some of the basics ourselves in a later chapter. Unfortunately, though, no matter how smart the algorithm, external fragmentation will still exist; thus, a good algorithm simply attempts to minimize it.

TIP: IF 1000 SOLUTIONS EXIST, NO GREAT ONE DOES
The fact that so many different algorithms exist to try to minimize external fragmentation is indicative of a stronger underlying truth: there is no one "best" way to solve the problem. Thus, we settle for something reasonable and hope it is good enough. The only real solution (as we will see in forthcoming chapters) is to avoid the problem altogether, by never allocating memory in variable-sized chunks.

16.7 Summary

Segmentation solves a number of problems, and helps us build a more effective virtualization of memory. Beyond just dynamic relocation, segmentation can better support sparse address spaces, by avoiding the huge potential waste of memory between logical segments of the address space. It is also fast, as doing the arithmetic segmentation requires is easy and well-suited to hardware; the overheads of translation are minimal. A fringe benefit arises too: code sharing. If code is placed within a separate segment, such a segment could potentially be shared across multiple running programs.

However, as we learned, allocating variable-sized segments in memory leads to some problems that we'd like to overcome. The first, as discussed above, is external fragmentation. Because segments are variable-sized, free memory gets chopped up into odd-sized pieces, and thus satisfying a memory-allocation request can be difficult. One can try to use smart algorithms [W+95] or periodically compact memory, but the problem is fundamental and hard to avoid.

The second and perhaps more important problem is that segmentation still isn't flexible enough to support our fully generalized, sparse address space. For example, if we have a large but sparsely-used heap all in one logical segment, the entire heap must still reside in memory in order to be accessed. In other words, if our model of how the address space is being used doesn't exactly match how the underlying segmentation has been designed to support it, segmentation doesn't work very well. We thus need to find some new solutions. Ready to find them?

References

[CV65] "Introduction and Overview of the Multics System"
F. J. Corbato and V. A. Vyssotsky
Fall Joint Computer Conference, 1965
One of five papers presented on Multics at the Fall Joint Computer Conference; oh to be a fly on the wall in that room that day!

[DD68] "Virtual Memory, Processes, and Sharing in Multics"
Robert C. Daley and Jack B. Dennis
Communications of the ACM, Volume 11, Issue 5, May 1968
An early paper on how to perform dynamic linking in Multics, which was way ahead of its time. Dynamic linking finally found its way back into systems about 20 years later, as the large X-windows libraries demanded it. Some say that these large X11 libraries were MIT's revenge for removing support for dynamic linking in early versions of UNIX*!*

[G62] "Fact Segmentation"
M. N. Greenfield
Proceedings of the SJCC, Volume 21, May 1962
Another early paper on segmentation; so early that it has no references to other work.

[H61] "Program Organization and Record Keeping for Dynamic Storage"
A. W. Holt
Communications of the ACM, Volume 4, Issue 10, October 1961
An incredibly early and difficult to read paper about segmentation and some of its uses.

[I09] "Intel 64 and IA-32 Architectures Software Developer's Manuals"
Intel, 2009
Available: http://www.intel.com/products/processor/manuals
Try reading about segmentation in here (Chapter 3 in Volume 3a); it'll hurt your head, at least a little bit.

[K68] "The Art of Computer Programming: Volume I"
Donald Knuth
Addison-Wesley, 1968
Knuth is famous not only for his early books on the Art of Computer Programming but for his typesetting system TeX which is still a powerhouse typesetting tool used by professionals today, and indeed to typeset this very book. His tomes on algorithms are a great early reference to many of the algorithms that underly computing systems today.

[L83] "Hints for Computer Systems Design"
Butler Lampson
ACM Operating Systems Review, 15:5, October 1983
A treasure-trove of sage advice on how to build systems. Hard to read in one sitting; take it in a little at a time, like a fine wine, or a reference manual.

[LL82] "Virtual Memory Management in the VAX/VMS Operating System"
Henry M. Levy and Peter H. Lipman
IEEE Computer, Volume 15, Number 3 (March 1982)
A classic memory management system, with lots of common sense in its design. We'll study it in more detail in a later chapter.

[RK68] "Dynamic Storage Allocation Systems"
B. Randell and C.J. Kuehner
Communications of the ACM
Volume 11(5), pages 297-306, May 1968
A nice overview of the differences between paging and segmentation, with some historical discussion of various machines.

[R69] "A note on storage fragmentation and program segmentation"
Brian Randell
Communications of the ACM
Volume 12(7), pages 365-372, July 1969
One of the earliest papers to discuss fragmentation.

[W+95] "Dynamic Storage Allocation: A Survey and Critical Review"
Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles
In International Workshop on Memory Management
Scotland, United Kingdom, September 1995
A great survey paper on memory allocators.

Homework

This program allows you to see how address translations are performed in a system with segmentation. See the README for details.

Questions

1. First let's use a tiny address space to translate some addresses. Here's a simple set of parameters with a few different random seeds; can you translate the addresses?

```
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2
```

2. Now, let's see if we understand this tiny address space we've constructed (using the parameters from the question above). What is the highest legal virtual address in segment 0? What about the lowest legal virtual address in segment 1? What are the lowest and highest *illegal* addresses in this entire address space? Finally, how would you run `segmentation.py` with the `-A` flag to test if you are right?
3. Let's say we have a tiny 16-byte address space in a 128-byte physical memory. What base and bounds would you set up so as to get the simulator to generate the following translation results for the specified address stream: valid, valid, violation, ..., violation, valid, valid? Assume the following parameters:

```
segmentation.py -a 16 -p 128
  -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
  --b0 ? --l0 ? --b1 ? --l1 ?
```

4. Assuming we want to generate a problem where roughly 90% of the randomly-generated virtual addresses are valid (i.e., not segmentation violations). How should you configure the simulator to do so? Which parameters are important?
5. Can you run the simulator such that no virtual addresses are valid? How?

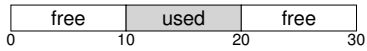
17

Free-Space Management

In this chapter, we take a small detour from our discussion of virtualizing memory to discuss a fundamental aspect of any memory management system, whether it be a malloc library (managing pages of a process's heap) or the OS itself (managing portions of the address space of a process). Specifically, we will discuss the issues surrounding **free-space management**.

Let us make the problem more specific. Managing free space can certainly be easy, as we will see when we discuss the concept of **paging**. It is easy when the space you are managing is divided into fixed-sized units; in such a case, you just keep a list of these fixed-sized units; when a client requests one of them, return the first entry.

Where free-space management becomes more difficult (and interesting) is when the free space you are managing consists of variable-sized units; this arises in a user-level memory-allocation library (as in `malloc()` and `free()`) and in an OS managing physical memory when using **segmentation** to implement virtual memory. In either case, the problem that exists is known as **external fragmentation**: the free space gets chopped into little pieces of different sizes and is thus fragmented; subsequent requests may fail because there is no single contiguous space that can satisfy the request, even though the total amount of free space exceeds the size of the request.



The figure shows an example of this problem. In this case, the total free space available is 20 bytes; unfortunately, it is fragmented into two chunks of size 10 each. As a result, a request for 15 bytes will fail even though there are 20 bytes free. And thus we arrive at the problem addressed in this chapter.

CRUX: HOW TO MANAGE FREE SPACE
How should free space be managed, when satisfying variable-sized requests? What strategies can be used to minimize fragmentation? What are the time and space overheads of alternate approaches?

17.1 Assumptions

Most of this discussion will focus on the great history of allocators found in user-level memory-allocation libraries. We draw on Wilson's excellent survey [W+95] but encourage interested readers to go to the source document itself for more details[1].

We assume a basic interface such as that provided by `malloc()` and `free()`. Specifically, `void *malloc(size_t size)` takes a single parameter, `size`, which is the number of bytes requested by the application; it hands back a pointer (of no particular type, or a **void pointer** in C lingo) to a region of that size (or greater). The complementary routine `void free(void *ptr)` takes a pointer and frees the corresponding chunk. Note the implication of the interface: the user, when freeing the space, does not inform the library of its size; thus, the library must be able to figure out how big a chunk of memory is when handed just a pointer to it. We'll discuss how to do this a bit later on in the chapter.

The space that this library manages is known historically as the `heap`, and the generic data structure used to manage free space in the heap is some kind of **free list**. This structure contains references to all of the free chunks of space in the managed region of memory. Of course, this data structure need not be a list *per se*, but just some kind of data structure to track free space.

We further assume that primarily we are concerned with **external fragmentation**, as described above. Allocators could of course also have the problem of **internal fragmentation**; if an allocator hands out chunks of memory bigger than that requested, any unasked for (and thus unused) space in such a chunk is considered *internal* fragmentation (because the waste occurs inside the allocated unit) and is another example of space waste. However, for the sake of simplicity, and because it is the more interesting of the two types of fragmentation, we'll mostly focus on external fragmentation.

We'll also assume that once memory is handed out to a client, it cannot be relocated to another location in memory. For example, if a program calls `malloc()` and is given a pointer to some space within the heap, that memory region is essentially "owned" by the program (and cannot be moved by the library) until the program returns it via a corresponding call to `free()`. Thus, no **compaction** of free space is possible, which

[1] It is nearly 80 pages long; thus, you really have to be interested!

would be useful to combat fragmentation[2]. Compaction could, however, be used in the OS to deal with fragmentation when implementing **segmentation** (as discussed in said chapter on segmentation).

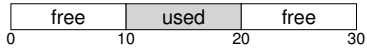
Finally, we'll assume that the allocator manages a contiguous region of bytes. In some cases, an allocator could ask for that region to grow; for example, a user-level memory-allocation library might call into the kernel to grow the heap (via a system call such as `sbrk`) when it runs out of space. However, for simplicity, we'll just assume that the region is a single fixed size throughout its life.

17.2 Low-level Mechanisms

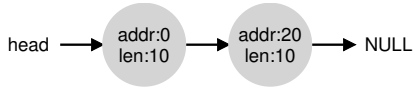
Before delving into some policy details, we'll first cover some common mechanisms used in most allocators. First, we'll discuss the basics of splitting and coalescing, common techniques in most any allocator. Second, we'll show how one can track the size of allocated regions quickly and with relative ease. Finally, we'll discuss how to build a simple list inside the free space to keep track of what is free and what isn't.

Splitting and Coalescing

A free list contains a set of elements that describe the free space still remaining in the heap. Thus, assume the following 30-byte heap:



The free list for this heap would have two elements on it. One entry describes the first 10-byte free segment (bytes 0-9), and one entry describes the other free segment (bytes 20-29):



As described above, a request for anything greater than 10 bytes will fail (returning NULL); there just isn't a single contiguous chunk of memory of that size available. A request for exactly that size (10 bytes) could be satisfied easily by either of the free chunks. But what happens if the request is for something *smaller* than 10 bytes?

Assume we have a request for just a single byte of memory. In this case, the allocator will perform an action known as **splitting**: it will find

[2]Once you hand a pointer to a chunk of memory to a C program, it is generally difficult to determine all references (pointers) to that region, which may be stored in other variables or even in registers at a given point in execution. This may not be the case in more strongly-typed, garbage-collected languages, which would thus enable compaction as a technique to combat fragmentation.

a free chunk of memory that can satisfy the request and split it into two. The first chunk it will return to the caller; the second chunk will remain on the list. Thus, in our example above, if a request for 1 byte were made, and the allocator decided to use the second of the two elements on the list to satisfy the request, the call to malloc() would return 20 (the address of the 1-byte allocated region) and the list would end up looking like this:

In the picture, you can see the list basically stays intact; the only change is that the free region now starts at 21 instead of 20, and the length of that free region is now just 9[3]. Thus, the split is commonly used in allocators when requests are smaller than the size of any particular free chunk.

A corollary mechanism found in many allocators is known as **coalescing** of free space. Take our example from above once more (free 10 bytes, used 10 bytes, and another free 10 bytes).

Given this (tiny) heap, what happens when an application calls free(10), thus returning the space in the middle of the heap? If we simply add this free space back into our list without too much thinking, we might end up with a list that looks like this:

Note the problem: while the entire heap is now free, it is seemingly divided into three chunks of 10 bytes each. Thus, if a user requests 20 bytes, a simple list traversal will not find such a free chunk, and return failure.

What allocators do in order to avoid this problem is coalesce free space when a chunk of memory is freed. The idea is simple: when returning a free chunk in memory, look carefully at the addresses of the chunk you are returning as well as the nearby chunks of free space; if the newly-freed space sits right next to one (or two, as in this example) existing free chunks, merge them into a single larger free chunk. Thus, with coalescing, our final list should look like this:

Indeed, this is what the heap list looked like at first, before any allocations were made. With coalescing, an allocator can better ensure that large free extents are available for the application.

[3] This discussion assumes that there are no headers, an unrealistic but simplifying assumption we make for now.

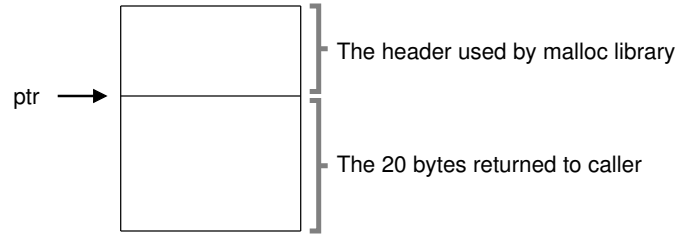


Figure 17.1: **An Allocated Region Plus Header**

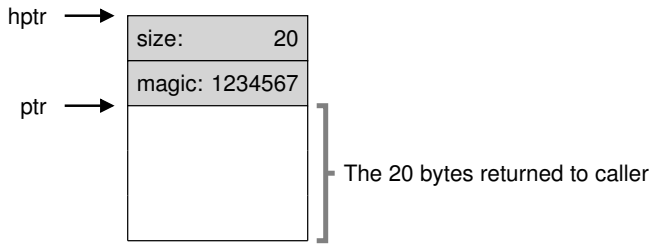


Figure 17.2: **Specific Contents Of The Header**

Tracking The Size Of Allocated Regions

You might have noticed that the interface to `free(void *ptr)` does not take a size parameter; thus it is assumed that given a pointer, the malloc library can quickly determine the size of the region of memory being freed and thus incorporate the space back into the free list.

To accomplish this task, most allocators store a little bit of extra information in a **header** block which is kept in memory, usually just before the handed-out chunk of memory. Let's look at an example again (Figure 17.1). In this example, we are examining an allocated block of size 20 bytes, pointed to by `ptr`; imagine the user called `malloc()` and stored the results in `ptr`, e.g., `ptr = malloc(20);`.

The header minimally contains the size of the allocated region (in this case, 20); it may also contain additional pointers to speed up deallocation, a magic number to provide additional integrity checking, and other information. Let's assume a simple header which contains the size of the region and a magic number, like this:

```
typedef struct __header_t {
    int size;
    int magic;
} header_t;
```

The example above would look like what you see in Figure 17.2. When

the user calls `free(ptr)`, the library then uses simple pointer arithmetic to figure out where the header begins:

```
void free(void *ptr) {
    header_t *hptr = (void *)ptr - sizeof(header_t);
    ...
```

After obtaining such a pointer to the header, the library can easily determine whether the magic number matches the expected value as a sanity check (`assert(hptr->magic == 1234567)`) and calculate the total size of the newly-freed region via simple math (i.e., adding the size of the header to size of the region). Note the small but critical detail in the last sentence: the size of the free region is the size of the header plus the size of the space allocated to the user. Thus, when a user requests N bytes of memory, the library does not search for a free chunk of size N; rather, it searches for a free chunk of size N plus the size of the header.

Embedding A Free List

Thus far we have treated our simple free list as a conceptual entity; it is just a list describing the free chunks of memory in the heap. But how do we build such a list inside the free space itself?

In a more typical list, when allocating a new node, you would just call `malloc()` when you need space for the node. Unfortunately, within the memory-allocation library, you can't do this! Instead, you need to build the list *inside* the free space itself. Don't worry if this sounds a little weird; it is, but not so weird that you can't do it!

Assume we have a 4096-byte chunk of memory to manage (i.e., the heap is 4KB). To manage this as a free list, we first have to initialize said list; initially, the list should have one entry, of size 4096 (minus the header size). Here is the description of a node of the list:

```
typedef struct __node_t {
    int              size;
    struct __node_t *next;
} node_t;
```

Now let's look at some code that initializes the heap and puts the first element of the free list inside that space. We are assuming that the heap is built within some free space acquired via a call to the system call `mmap()`; this is not the only way to build such a heap but serves us well in this example. Here is the code:

```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                    MAP_ANON|MAP_PRIVATE, -1, 0);
head->size   = 4096 - sizeof(node_t);
head->next   = NULL;
```

After running this code, the status of the list is that it has a single entry, of size 4088. Yes, this is a tiny heap, but it serves as a fine example for us

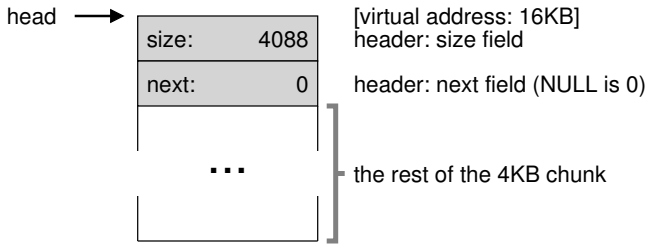


Figure 17.3: **A Heap With One Free Chunk**

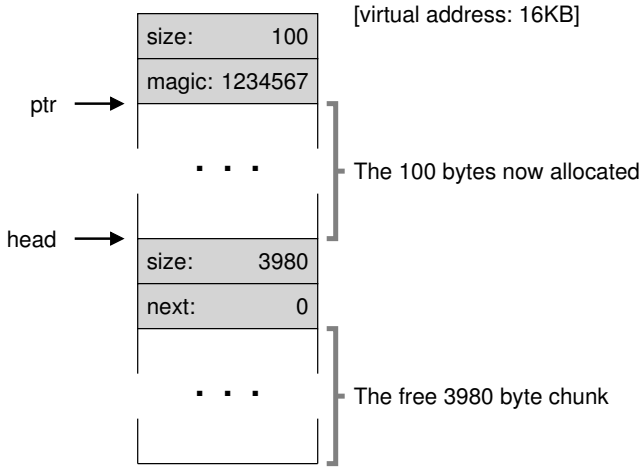


Figure 17.4: **A Heap: After One Allocation**

here. The `head` pointer contains the beginning address of this range; let's assume it is 16KB (though any virtual address would be fine). Visually, the heap thus looks like what you see in Figure 17.3.

Now, let's imagine that a chunk of memory is requested, say of size 100 bytes. To service this request, the library will first find a chunk that is large enough to accommodate the request; because there is only one free chunk (size: 4088), this chunk will be chosen. Then, the chunk will be **split** into two: one chunk big enough to service the request (and header, as described above), and the remaining free chunk. Assuming an 8-byte header (an integer size and an integer magic number), the space in the heap now looks like what you see in Figure 17.4.

Thus, upon the request for 100 bytes, the library allocated 108 bytes out of the existing one free chunk, returns a pointer (marked `ptr` in the figure above) to it, stashes the header information immediately before the

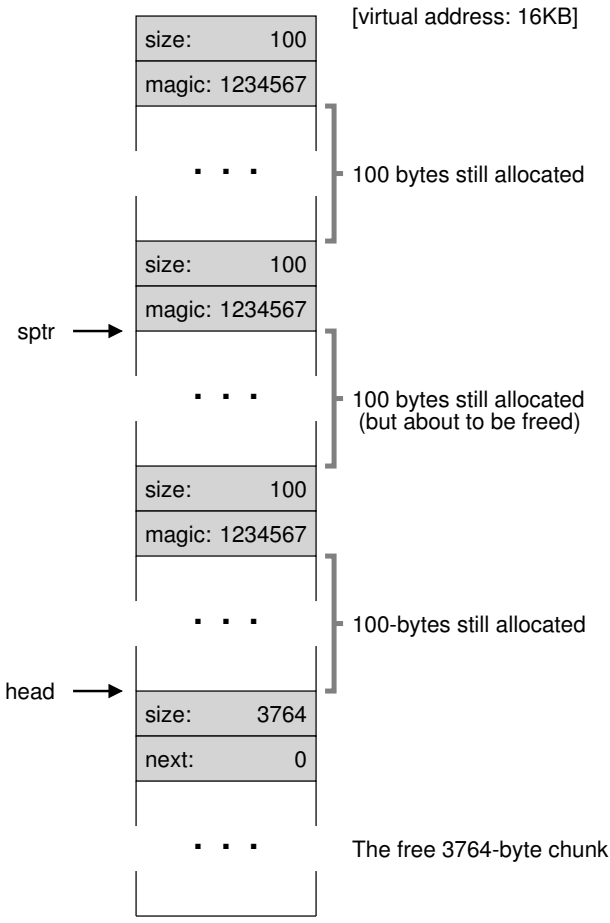


Figure 17.5: **Free Space With Three Chunks Allocated**

allocated space for later use upon `free()`, and shrinks the one free node in the list to 3980 bytes (4088 minus 108).

Now let's look at the heap when there are three allocated regions, each of 100 bytes (or 108 including the header). A visualization of this heap is shown in Figure 17.5.

As you can see therein, the first 324 bytes of the heap are now allocated, and thus we see three headers in that space as well as three 100-byte regions being used by the calling program. The free list remains uninteresting: just a single node (pointed to by `head`), but now only 3764 bytes in size after the three splits. But what happens when the calling program returns some memory via `free()`?

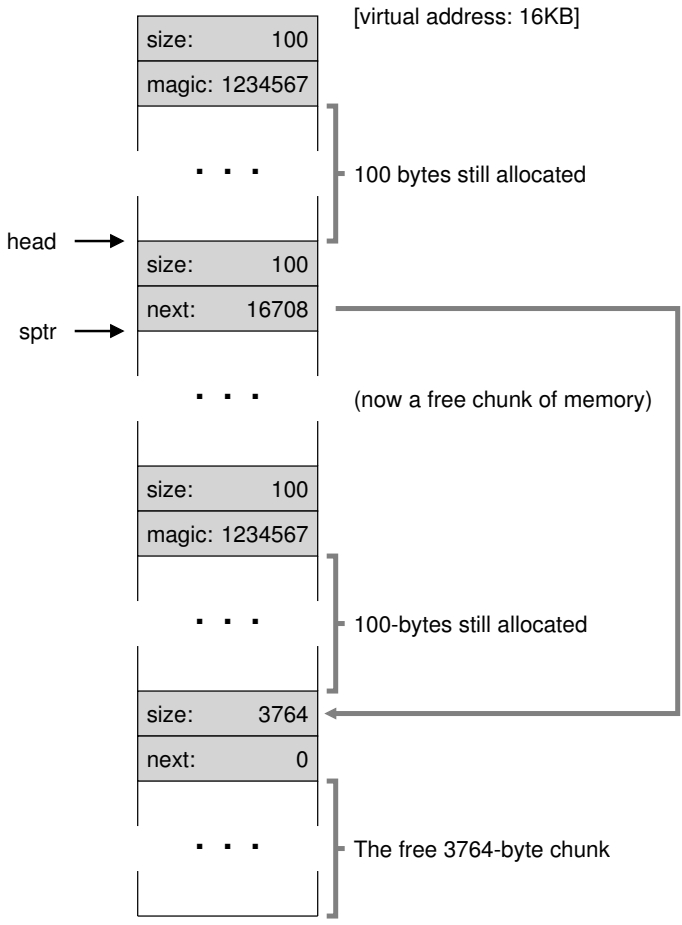


Figure 17.6: **Free Space With Two Chunks Allocated**

In this example, the application returns the middle chunk of allocated memory, by calling `free(16500)` (the value 16500 is arrived upon by adding the start of the memory region, 16384, to the 108 of the previous chunk and the 8 bytes of the header for this chunk). This value is shown in the previous diagram by the pointer `sptr`.

The library immediately figures out the size of the free region, and then adds the free chunk back onto the free list. Assuming we insert at the head of the free list, the space now looks like this (Figure 17.6).

And now we have a list that starts with a small free chunk (100 bytes, pointed to by the head of the list) and a large free chunk (3764 bytes).

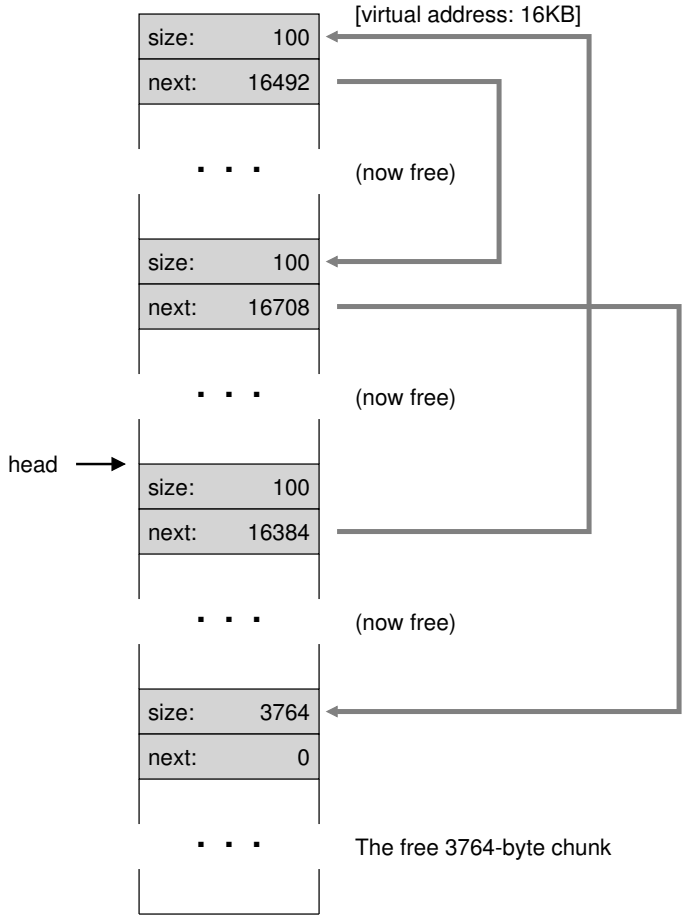


Figure 17.7: **A Non-Coalesced Free List**

Our list finally has more than one element on it! And yes, the free space is fragmented, an unfortunate but common occurrence.

One last example: let's assume now that the last two in-use chunks are freed. Without coalescing, you might end up with a free list that is highly fragmented (see Figure 17.7).

As you can see from the figure, we now have a big mess! Why? Simple, we forgot to **coalesce** the list. Although all of the memory is free, it is chopped up into pieces, thus appearing as a fragmented memory despite not being one. The solution is simple: go through the list and **merge** neighboring chunks; when finished, the heap will be whole again.

Growing The Heap

We should discuss one last mechanism found within many allocation libraries. Specifically, what should you do if the heap runs out of space? The simplest approach is just to fail. In some cases this is the only option, and thus returning NULL is an honorable approach. Don't feel bad! You tried, and though you failed, you fought the good fight.

Most traditional allocators start with a small-sized heap and then request more memory from the OS when they run out. Typically, this means they make some kind of system call (e.g., `sbrk` in most UNIX systems) to grow the heap, and then allocate the new chunks from there. To service the `sbrk` request, the OS finds free physical pages, maps them into the address space of the requesting process, and then returns the value of the end of the new heap; at that point, a larger heap is available, and the request can be successfully serviced.

17.3 Basic Strategies

Now that we have some machinery under our belt, let's go over some basic strategies for managing free space. These approaches are mostly based on pretty simple policies that you could think up yourself; try it before reading and see if you come up with all of the alternatives (or maybe some new ones!).

The ideal allocator is both fast and minimizes fragmentation. Unfortunately, because the stream of allocation and free requests can be arbitrary (after all, they are determined by the programmer), any particular strategy can do quite badly given the wrong set of inputs. Thus, we will not describe a "best" approach, but rather talk about some basics and discuss their pros and cons.

Best Fit

The **best fit** strategy is quite simple: first, search through the free list and find chunks of free memory that are as big or bigger than the requested size. Then, return the one that is the smallest in that group of candidates; this is the so called best-fit chunk (it could be called smallest fit too). One pass through the free list is enough to find the correct block to return.

The intuition behind best fit is simple: by returning a block that is close to what the user asks, best fit tries to reduce wasted space. However, there is a cost; naive implementations pay a heavy performance penalty when performing an exhaustive search for the correct free block.

Worst Fit

The **worst fit** approach is the opposite of best fit; find the largest chunk and return the requested amount; keep the remaining (large) chunk on the free list. Worst fit tries to thus leave big chunks free instead of lots of

small chunks that can arise from a best-fit approach. Once again, however, a full search of free space is required, and thus this approach can be costly. Worse, most studies show that it performs badly, leading to excess fragmentation while still having high overheads.

First Fit

The **first fit** method simply finds the first block that is big enough and returns the requested amount to the user. As before, the remaining free space is kept free for subsequent requests.

First fit has the advantage of speed — no exhaustive search of all the free spaces are necessary — but sometimes pollutes the beginning of the free list with small objects. Thus, how the allocator manages the free list's order becomes an issue. One approach is to use **address-based ordering**; by keeping the list ordered by the address of the free space, coalescing becomes easier, and fragmentation tends to be reduced.

Next Fit

Instead of always beginning the first-fit search at the beginning of the list, the **next fit** algorithm keeps an extra pointer to the location within the list where one was looking last. The idea is to spread the searches for free space throughout the list more uniformly, thus avoiding splintering of the beginning of the list. The performance of such an approach is quite similar to first fit, as an exhaustive search is once again avoided.

Examples

Here are a few examples of the above strategies. Envision a free list with three elements on it, of sizes 10, 30, and 20 (we'll ignore headers and other details here, instead just focusing on how strategies operate):

Assume an allocation request of size 15. A best-fit approach would search the entire list and find that 20 was the best fit, as it is the smallest free space that can accommodate the request. The resulting free list:

As happens in this example, and often happens with a best-fit approach, a small free chunk is now left over. A worst-fit approach is similar but instead finds the largest chunk, in this example 30. The resulting list:

The first-fit strategy, in this example, does the same thing as worst-fit, also finding the first free block that can satisfy the request. The difference is in the search cost; both best-fit and worst-fit look through the entire list; first-fit only examines free chunks until it finds one that fits, thus reducing search cost.

These examples just scratch the surface of allocation policies. More detailed analysis with real workloads and more complex allocator behaviors (e.g., coalescing) are required for a deeper understanding. Perhaps something for a homework section, you say?

17.4 Other Approaches

Beyond the basic approaches described above, there have been a host of suggested techniques and algorithms to improve memory allocation in some way. We list a few of them here for your consideration (i.e., to make you think about a little more than just best-fit allocation).

Segregated Lists

One interesting approach that has been around for some time is the use of **segregated lists**. The basic idea is simple: if a particular application has one (or a few) popular-sized request that it makes, keep a separate list just to manage objects of that size; all other requests are forwarded to a more general memory allocator.

The benefits of such an approach are obvious. By having a chunk of memory dedicated for one particular size of requests, fragmentation is much less of a concern; moreover, allocation and free requests can be served quite quickly when they are of the right size, as no complicated search of a list is required.

Just like any good idea, this approach introduces new complications into a system as well. For example, how much memory should one dedicate to the pool of memory that serves specialized requests of a given size, as opposed to the general pool? One particular allocator, the **slab allocator** by uber-engineer Jeff Bonwick (which was designed for use in the Solaris kernel), handles this issue in a rather nice way [B94].

Specifically, when the kernel boots up, it allocates a number of **object caches** for kernel objects that are likely to be requested frequently (such as locks, file-system inodes, etc.); the object caches thus are each segregated free lists of a given size and serve memory allocation and free requests quickly. When a given cache is running low on free space, it requests some **slabs** of memory from a more general memory allocator (the total amount requested being a multiple of the page size and the object in question). Conversely, when the reference counts of the objects within a given slab all go to zero, the general allocator can reclaim them from the specialized allocator, which is often done when the VM system needs more memory.

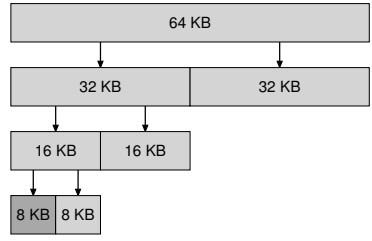
ASIDE: **GREAT ENGINEERS ARE REALLY GREAT**
Engineers like Jeff Bonwick (who not only wrote the slab allocator mentioned herein but also was the lead of an amazing file system, ZFS) are the heart of Silicon Valley. Behind almost any great product or technology is a human (or small group of humans) who are way above average in their talents, abilities, and dedication. As Mark Zuckerberg (of Facebook) says: "Someone who is exceptional in their role is not just a little better than someone who is pretty good. They are 100 times better." This is why, still today, one or two people can start a company that changes the face of the world forever (think Google, Apple, or Facebook). Work hard and you might become such a "100x" person as well. Failing that, work *with* such a person; you'll learn more in a day than most learn in a month. Failing that, feel sad.

The slab allocator also goes beyond most segregated list approaches by keeping free objects on the lists in a pre-initialized state. Bonwick shows that initialization and destruction of data structures is costly [B94]; by keeping freed objects in a particular list in their initialized state, the slab allocator thus avoids frequent initialization and destruction cycles per object and thus lowers overheads noticeably.

Buddy Allocation

Because coalescing is critical for an allocator, some approaches have been designed around making coalescing simple. One good example is found in the **binary buddy allocator** [K65].

In such a system, free memory is first conceptually thought of as one big space of size 2^N. When a request for memory is made, the search for free space recursively divides free space by two until a block that is big enough to accommodate the request is found (and a further split into two would result in a space that is too small). At this point, the requested block is returned to the user. Here is an example of a 64KB free space getting divided in the search for a 7KB block:



In the example, the leftmost 8KB block is allocated (as indicated by the darker shade of gray) and returned to the user; note that this scheme can suffer from **internal fragmentation**, as you are only allowed to give out power-of-two-sized blocks.

The beauty of buddy allocation is found in what happens when that block is freed. When returning the 8KB block to the free list, the allocator checks whether the "buddy" 8KB is free; if so, it coalesces the two blocks into a 16KB block. The allocator then checks if the buddy of the 16KB block is still free; if so, it coalesces those two blocks. This recursive coalescing process continues up the tree, either restoring the entire free space or stopping when a buddy is found to be in use.

The reason buddy allocation works so well is that it is simple to determine the buddy of a particular block. How, you ask? Think about the addresses of the blocks in the free space above. If you think carefully enough, you'll see that the address of each buddy pair only differs by a single bit; which bit is determined by the level in the buddy tree. And thus you have a basic idea of how binary buddy allocation schemes work. For more detail, as always, see the Wilson survey [W+95].

Other Ideas

One major problem with many of the approaches described above is their lack of **scaling**. Specifically, searching lists can be quite slow. Thus, advanced allocators use more complex data structures to address these costs, trading simplicity for performance. Examples include balanced binary trees, splay trees, or partially-ordered trees [W+95].

Given that modern systems often have multiple processors and run multi-threaded workloads (something you'll learn about in great detail in the section of the book on Concurrency), it is not surprising that a lot of effort has been spent making allocators work well on multiprocessor-based systems. Two wonderful examples are found in Berger et al. [B+00] and Evans [E06]; check them out for the details.

These are but two of the thousands of ideas people have had over time about memory allocators; read on your own if you are curious. Failing that, read about how the glibc allocator works [S15], to give you a sense of what the real world is like.

17.5 Summary

In this chapter, we've discussed the most rudimentary forms of memory allocators. Such allocators exist everywhere, linked into every C program you write, as well as in the underlying OS which is managing memory for its own data structures. As with many systems, there are many trade-offs to be made in building such a system, and the more you know about the exact workload presented to an allocator, the more you could do to tune it to work better for that workload. Making a fast, space-efficient, scalable allocator that works well for a broad range of workloads remains an on-going challenge in modern computer systems.

References

[B+00] "Hoard: A Scalable Memory Allocator for Multithreaded Applications"
Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson
ASPLOS-IX, November 2000
Berger and company's excellent allocator for multiprocessor systems. Beyond just being a fun paper, also used in practice!

[B94] "The Slab Allocator: An Object-Caching Kernel Memory Allocator"
Jeff Bonwick
USENIX '94
A cool paper about how to build an allocator for an operating system kernel, and a great example of how to specialize for particular common object sizes.

[E06] "A Scalable Concurrent malloc(3) Implementation for FreeBSD"
Jason Evans
http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf
April 2006
A detailed look at how to build a real modern allocator for use in multiprocessors. The "jemalloc" allocator is in widespread use today, within FreeBSD, NetBSD, Mozilla Firefox, and within Facebook.

[K65] "A Fast Storage Allocator"
Kenneth C. Knowlton
Communications of the ACM, Volume 8, Number 10, October 1965
The common reference for buddy allocation. Random strange fact: Knuth gives credit for the idea not to Knowlton but to Harry Markowitz, a Nobel-prize winning economist. Another strange fact: Knuth communicates all of his emails via a secretary; he doesn't send email himself, rather he tells his secretary what email to send and then the secretary does the work of emailing. Last Knuth fact: he created TeX, the tool used to typeset this book. It is an amazing piece of software[4].

[S15] "Understanding glibc malloc"
Sploitfun
February, 2015
https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/
A deep dive into how glibc malloc works. Amazingly detailed and a very cool read.

[W+95] "Dynamic Storage Allocation: A Survey and Critical Review"
Paul R. Wilson, Mark S. Johnstone, Michael Neely, David Boles
International Workshop on Memory Management
Kinross, Scotland, September 1995
An excellent and far-reaching survey of many facets of memory allocation. Far too much detail to go into in this tiny chapter!

[4] Actually we use LaTeX, which is based on Lamport's additions to TeX, but close enough.

Homework

The program, `malloc.py`, lets you explore the behavior of a simple free-space allocator as described in the chapter. See the README for details of its basic operation.

Questions

1. First run with the flags `-n 10 -H 0 -p BEST -s 0` to generate a few random allocations and frees. Can you predict what alloc()/free() will return? Can you guess the state of the free list after each request? What do you notice about the free list over time?
2. How are the results different when using a WORST fit policy to search the free list (`-p WORST`)? What changes?
3. What about when using FIRST fit (`-p FIRST`)? What speeds up when you use first fit?
4. For the above questions, how the list is kept ordered can affect the time it takes to find a free location for some of the policies. Use the different free list orderings (`-l ADDRSORT`, `-l SIZESORT+`, `-l SIZESORT-`) to see how the policies and the list orderings interact.
5. Coalescing of a free list can be quite important. Increase the number of random allocations (say to `-n 1000`). What happens to larger allocation requests over time? Run with and without coalescing (i.e., without and with the `-C` flag). What differences in outcome do you see? How big is the free list over time in each case? Does the ordering of the list matter in this case?
6. What happens when you change the percent allocated fraction `-P` to higher than 50? What happens to allocations as it nears 100? What about as it nears 0?
7. What kind of specific requests can you make to generate a highly-fragmented free space? Use the `-A` flag to create fragmented free lists, and see how different policies and options change the organization of the free list.

18

Paging: Introduction

It is sometimes said that the operating system takes one of two approaches when solving most any space-management problem. The first approach is to chop things up into *variable-sized* pieces, as we saw with **segmentation** in virtual memory. Unfortunately, this solution has inherent difficulties. In particular, when dividing a space into different-size chunks, the space itself can become **fragmented**, and thus allocation becomes more challenging over time.

Thus, it may be worth considering the second approach: to chop up space into *fixed-sized* pieces. In virtual memory, we call this idea **paging**, and it goes back to an early and important system, the Atlas [KE+62, L78]. Instead of splitting up a process's address space into some number of variable-sized logical segments (e.g., code, heap, stack), we divide it into fixed-sized units, each of which we call a **page**. Correspondingly, we view physical memory as an array of fixed-sized slots called **page frames**; each of these frames can contain a single virtual-memory page. Our challenge:

THE CRUX:
HOW TO VIRTUALIZE MEMORY WITH PAGES

How can we virtualize memory with pages, so as to avoid the problems of segmentation? What are the basic techniques? How do we make those techniques work well, with minimal space and time overheads?

18.1 A Simple Example And Overview

To help make this approach more clear, let's illustrate it with a simple example. Figure 18.1 (page 2) presents an example of a tiny address space, only 64 bytes total in size, with four 16-byte pages (virtual pages 0, 1, 2, and 3). Real address spaces are much bigger, of course, commonly 32 bits and thus 4-GB of address space, or even 64 bits[1]; in the book, we'll often use tiny examples to make them easier to digest.

[1] A 64-bit address space is hard to imagine, it is so amazingly large. An analogy might help: if you think of a 32-bit address space as the size of a tennis court, a 64-bit address space is about the size of Europe(!).

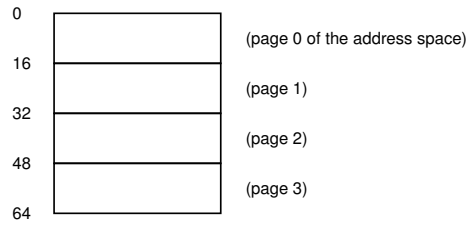


Figure 18.1: **A Simple 64-byte Address Space**

Physical memory, as shown in Figure 18.2, also consists of a number of fixed-sized slots, in this case eight page frames (making for a 128-byte physical memory, also ridiculously small). As you can see in the diagram, the pages of the virtual address space have been placed at different locations throughout physical memory; the diagram also shows the OS using some of physical memory for itself.

Paging, as we will see, has a number of advantages over our previous approaches. Probably the most important improvement will be *flexibility*: with a fully-developed paging approach, the system will be able to support the abstraction of an address space effectively, regardless of how a process uses the address space; we won't, for example, make assumptions about the direction the heap and stack grow and how they are used.

Another advantage is the *simplicity* of free-space management that paging affords. For example, when the OS wishes to place our tiny 64-byte address space into our eight-page physical memory, it simply finds four free pages; perhaps the OS keeps a **free list** of all free pages for this, and just grabs the first four free pages off of this list. In the example, the OS

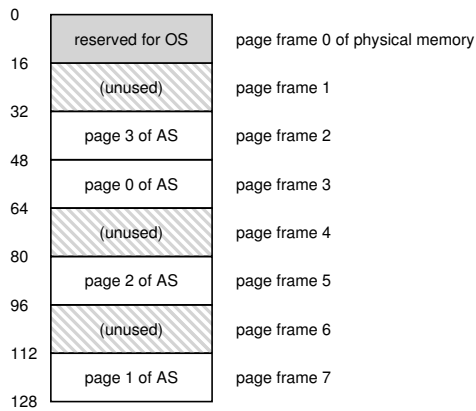


Figure 18.2: **A 64-Byte Address Space In A 128-Byte Physical Memory**

has placed virtual page 0 of the address space (AS) in physical frame 3, virtual page 1 of the AS in physical frame 7, page 2 in frame 5, and page 3 in frame 2. Page frames 1, 4, and 6 are currently free.

To record where each virtual page of the address space is placed in physical memory, the operating system usually keeps a *per-process* data structure known as a **page table**. The major role of the page table is to store **address translations** for each of the virtual pages of the address space, thus letting us know where in physical memory each page resides. For our simple example (Figure 18.2, page 2), the page table would thus have the following four entries: (Virtual Page 0 $\rightarrow$ Physical Frame 3), (VP 1 $\rightarrow$ PF 7), (VP 2 $\rightarrow$ PF 5), and (VP 3 $\rightarrow$ PF 2).

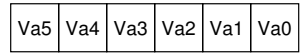
It is important to remember that this page table is a *per-process* data structure (most page table structures we discuss are per-process structures; an exception we'll touch on is the **inverted page table**). If another process were to run in our example above, the OS would have to manage a different page table for it, as its virtual pages obviously map to *different* physical pages (modulo any sharing going on).

Now, we know enough to perform an address-translation example. Let's imagine the process with that tiny address space (64 bytes) is performing a memory access:

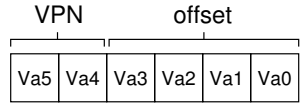
```
movl <virtual address>, %eax
```

Specifically, let's pay attention to the explicit load of the data from address `<virtual address>` into the register `eax` (and thus ignore the instruction fetch that must have happened prior).

To **translate** this virtual address that the process generated, we have to first split it into two components: the **virtual page number (VPN)**, and the **offset** within the page. For this example, because the virtual address space of the process is 64 bytes, we need 6 bits total for our virtual address ($2^6 = 64$). Thus, our virtual address can be conceptualized as follows:



In this diagram, Va5 is the highest-order bit of the virtual address, and Va0 the lowest-order bit. Because we know the page size (16 bytes), we can further divide the virtual address as follows:

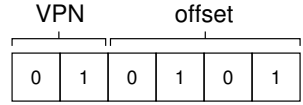


The page size is 16 bytes in a 64-byte address space; thus we need to be able to select 4 pages, and the top 2 bits of the address do just that. Thus, we have a 2-bit virtual page number (VPN). The remaining bits tell us which byte of the page we are interested in, 4 bits in this case; we call this the offset.

When a process generates a virtual address, the OS and hardware must combine to translate it into a meaningful physical address. For example, let us assume the load above was to virtual address 21:

```
movl 21, %eax
```

Turning "21" into binary form, we get "010101", and thus we can examine this virtual address and see how it breaks down into a virtual page number (VPN) and offset:



Thus, the virtual address "21" is on the 5th ("0101"th) byte of virtual page "01" (or 1). With our virtual page number, we can now index our page table and find which physical frame virtual page 1 resides within. In the page table above the **physical frame number (PFN)** (also sometimes called the **physical page number** or **PPN**) is 7 (binary 111). Thus, we can translate this virtual address by replacing the VPN with the PFN and then issue the load to physical memory (Figure 18.3).

Note the offset stays the same (i.e., it is not translated), because the offset just tells us which byte *within* the page we want. Our final physical address is 1110101 (117 in decimal), and is exactly where we want our load to fetch data from (Figure 18.2, page 2).

With this basic overview in mind, we can now ask (and hopefully, answer) a few basic questions you may have about paging. For example, where are these page tables stored? What are the typical contents of the page table, and how big are the tables? Does paging make the system (too) slow? These and other beguiling questions are answered, at least in part, in the text below. Read on!

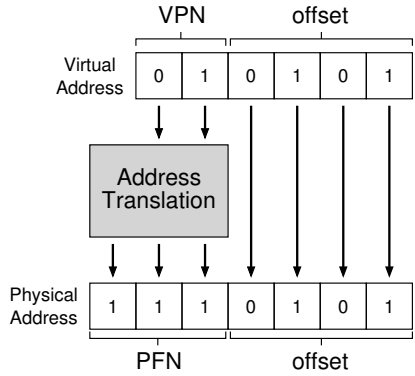


Figure 18.3: **The Address Translation Process**

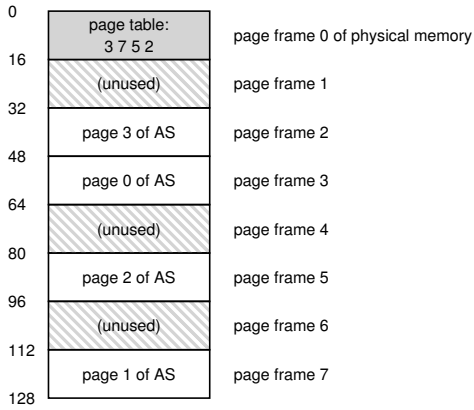


Figure 18.4: **Example: Page Table in Kernel Physical Memory**

18.2 Where Are Page Tables Stored?

Page tables can get terribly large, much bigger than the small segment table or base/bounds pair we have discussed previously. For example, imagine a typical 32-bit address space, with 4KB pages. This virtual address splits into a 20-bit VPN and 12-bit offset (recall that 10 bits would be needed for a 1KB page size, and just add two more to get to 4KB).

A 20-bit VPN implies that there are 2^{20} translations that the OS would have to manage for each process (that's roughly a million); assuming we need 4 bytes per **page table entry (PTE)** to hold the physical translation plus any other useful stuff, we get an immense 4MB of memory needed for each page table! That is pretty large. Now imagine there are 100 processes running: this means the OS would need 400MB of memory just for all those address translations! Even in the modern era, where machines have gigabytes of memory, it seems a little crazy to use a large chunk of it just for translations, no? And we won't even think about how big such a page table would be for a 64-bit address space; that would be too gruesome and perhaps scare you off entirely.

Because page tables are so big, we don't keep any special on-chip hardware in the MMU to store the page table of the currently-running process. Instead, we store the page table for each process in *memory* somewhere. Let's assume for now that the page tables live in physical memory that the OS manages; later we'll see that much of OS memory itself can be virtualized, and thus page tables can be stored in OS virtual memory (and even swapped to disk), but that is too confusing right now, so we'll ignore it. In Figure 18.4 is a picture of a page table in OS memory; see the tiny set of translations in there?

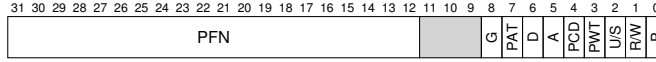


Figure 18.5: **An x86 Page Table Entry (PTE)**

18.3 What's Actually In The Page Table?

Let's talk a little about page table organization. The page table is just a data structure that is used to map virtual addresses (or really, virtual page numbers) to physical addresses (physical frame numbers). Thus, any data structure could work. The simplest form is called a **linear page table**, which is just an array. The OS *indexes* the array by the virtual page number (VPN), and looks up the page-table entry (PTE) at that index in order to find the desired physical frame number (PFN). For now, we will assume this simple linear structure; in later chapters, we will make use of more advanced data structures to help solve some problems with paging.

As for the contents of each PTE, we have a number of different bits in there worth understanding at some level. A **valid bit** is common to indicate whether the particular translation is valid; for example, when a program starts running, it will have code and heap at one end of its address space, and the stack at the other. All the unused space in-between will be marked **invalid**, and if the process tries to access such memory, it will generate a trap to the OS which will likely terminate the process. Thus, the valid bit is crucial for supporting a sparse address space; by simply marking all the unused pages in the address space invalid, we remove the need to allocate physical frames for those pages and thus save a great deal of memory.

We also might have **protection bits**, indicating whether the page could be read from, written to, or executed from. Again, accessing a page in a way not allowed by these bits will generate a trap to the OS.

There are a couple of other bits that are important but we won't talk about much for now. A **present bit** indicates whether this page is in physical memory or on disk (i.e., it has been **swapped out**). We will understand this machinery further when we study how to **swap** parts of the address space to disk to support address spaces that are larger than physical memory; swapping allows the OS to free up physical memory by moving rarely-used pages to disk. A **dirty bit** is also common, indicating whether the page has been modified since it was brought into memory.

A **reference bit** (a.k.a. **accessed bit**) is sometimes used to track whether a page has been accessed, and is useful in determining which pages are popular and thus should be kept in memory; such knowledge is critical during **page replacement**, a topic we will study in great detail in subsequent chapters.

Figure 18.5 shows an example page table entry from the x86 architecture [I09]. It contains a present bit (P); a read/write bit (R/W) which determines if writes are allowed to this page; a user/supervisor bit (U/S)

which determines if user-mode processes can access the page; a few bits (PWT, PCD, PAT, and G) that determine how hardware caching works for these pages; an accessed bit (A) and a dirty bit (D); and finally, the page frame number (PFN) itself.

Read the Intel Architecture Manuals [I09] for more details on x86 paging support. Be forewarned, however; reading manuals such as these, while quite informative (and certainly necessary for those who write code to use such page tables in the OS), can be challenging at first. A little patience, and a lot of desire, is required.

18.4 Paging: Also Too Slow

With page tables in memory, we already know that they might be too big. As it turns out, they can slow things down too. For example, take our simple instruction:

```
movl 21, %eax
```

Again, let's just examine the explicit reference to address 21 and not worry about the instruction fetch. In this example, we'll assume the hardware performs the translation for us. To fetch the desired data, the system must first **translate** the virtual address (21) into the correct physical address (117). Thus, before fetching the data from address 117, the system must first fetch the proper page table entry from the process's page table, perform the translation, and then load the data from physical memory.

To do so, the hardware must know where the page table is for the currently-running process. Let's assume for now that a single **page-table base register** contains the physical address of the starting location of the page table. To find the location of the desired PTE, the hardware will thus perform the following functions:

```
VPN     = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))
```

In our example, `VPN_MASK` would be set to 0x30 (hex 30, or binary 110000) which picks out the VPN bits from the full virtual address; `SHIFT` is set to 4 (the number of bits in the offset), such that we move the VPN bits down to form the correct integer virtual page number. For example, with virtual address 21 (010101), and masking turns this value into 010000; the shift turns it into 01, or virtual page 1, as desired. We then use this value as an index into the array of PTEs pointed to by the page table base register.

Once this physical address is known, the hardware can fetch the PTE from memory, extract the PFN, and concatenate it with the offset from the virtual address to form the desired physical address. Specifically, you can think of the PFN being left-shifted by `SHIFT`, and then logically OR'd with the offset to form the final address as follows:

```
offset   = VirtualAddress & OFFSET_MASK
PhysAddr = (PFN << SHIFT) | offset
```

```
1  // Extract the VPN from the virtual address
2  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3
4  // Form the address of the page-table entry (PTE)
5  PTEAddr = PTBR + (VPN * sizeof(PTE))
6
7  // Fetch the PTE
8  PTE = AccessMemory(PTEAddr)
9
10 // Check if process can access the page
11 if (PTE.Valid == False)
12     RaiseException(SEGMENTATION_FAULT)
13 else if (CanAccess(PTE.ProtectBits) == False)
14     RaiseException(PROTECTION_FAULT)
15 else
16     // Access is OK: form physical address and fetch it
17     offset   = VirtualAddress & OFFSET_MASK
18     PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19     Register = AccessMemory(PhysAddr)
```

Figure 18.6: **Accessing Memory With Paging**

Finally, the hardware can fetch the desired data from memory and put it into register `eax`. The program has now succeeded at loading a value from memory!

To summarize, we now describe the initial protocol for what happens on each memory reference. Figure 18.6 shows the basic approach. For every memory reference (whether an instruction fetch or an explicit load or store), paging requires us to perform one extra memory reference in order to first fetch the translation from the page table. That is a lot of work! Extra memory references are costly, and in this case will likely slow down the process by a factor of two or more.

And now you can hopefully see that there are *two* real problems that we must solve. Without careful design of both hardware and software, page tables will cause the system to run too slowly, as well as take up too much memory. While seemingly a great solution for our memory virtualization needs, these two crucial problems must first be overcome.

18.5 A Memory Trace

Before closing, we now trace through a simple memory access example to demonstrate all of the resulting memory accesses that occur when using paging. The code snippet (in C, in a file called `array.c`) that we are interested in is as follows:

```
int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;
```

We compile `array.c` and run it with the following commands:

ASIDE: **DATA STRUCTURE — THE PAGE TABLE**
One of the most important data structures in the memory management subsystem of a modern OS is the **page table**. In general, a page table stores **virtual-to-physical address translations**, thus letting the system know where each page of an address space actually resides in physical memory. Because each address space requires such translations, in general there is one page table per process in the system. The exact structure of the page table is either determined by the hardware (older systems) or can be more flexibly managed by the OS (modern systems).

```
prompt> gcc -o array array.c -Wall -O
prompt> ./array
```

Of course, to truly understand what memory accesses this code snippet (which simply initializes an array) will make, we'll have to know (or assume) a few more things. First, we'll have to **disassemble** the resulting binary (using `objdump` on Linux, or `otool` on a Mac) to see what assembly instructions are used to initialize the array in a loop. Here is the resulting assembly code:

```
1024 movl $0x0,(%edi,%eax,4)
1028 incl %eax
1032 cmpl $0x03e8,%eax
1036 jne  0x1024
```

The code, if you know a little **x86**, is actually quite easy to understand[2]. The first instruction moves the value zero (shown as `$0x0`) into the virtual memory address of the location of the array; this address is computed by taking the contents of `%edi` and adding `%eax` multiplied by four to it. Thus, `%edi` holds the base address of the array, whereas `%eax` holds the array index (`i`); we multiply by four because the array is an array of integers, each of size four bytes.

The second instruction increments the array index held in `%eax`, and the third instruction compares the contents of that register to the hex value `0x03e8`, or decimal 1000. If the comparison shows that two values are not yet equal (which is what the `jne` instruction tests), the fourth instruction jumps back to the top of the loop.

To understand which memory accesses this instruction sequence makes (at both the virtual and physical levels), we'll have to assume something about where in virtual memory the code snippet and array are found, as well as the contents and location of the page table.

For this example, we assume a virtual address space of size 64KB (unrealistically small). We also assume a page size of 1KB.

[2]We are cheating a little bit here, assuming each instruction is four bytes in size for simplicity; in actuality, x86 instructions are variable-sized.

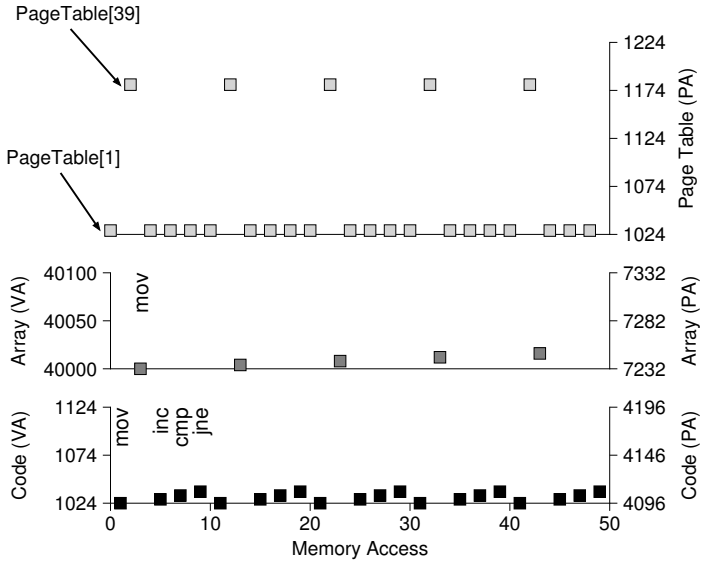


Figure 18.7: **A Virtual (And Physical) Memory Trace**

All we need to know now are the contents of the page table, and its location in physical memory. Let's assume we have a linear (array-based) page table and that it is located at physical address 1KB (1024).

As for its contents, there are just a few virtual pages we need to worry about having mapped for this example. First, there is the virtual page the code lives on. Because the page size is 1KB, virtual address 1024 resides on the second page of the virtual address space (VPN=1, as VPN=0 is the first page). Let's assume this virtual page maps to physical frame 4 (VPN 1 $\rightarrow$ PFN 4).

Next, there is the array itself. Its size is 4000 bytes (1000 integers), and we assume that it resides at virtual addresses 40000 through 44000 (not including the last byte). The virtual pages for this decimal range are VPN=39 ... VPN=42. Thus, we need mappings for these pages. Let's assume these virtual-to-physical mappings for the example: (VPN 39 $\rightarrow$ PFN 7), (VPN 40 $\rightarrow$ PFN 8), (VPN 41 $\rightarrow$ PFN 9), (VPN 42 $\rightarrow$ PFN 10).

We are now ready to trace the memory references of the program. When it runs, each instruction fetch will generate two memory references: one to the page table to find the physical frame that the instruction resides within, and one to the instruction itself to fetch it to the CPU for processing. In addition, there is one explicit memory reference in the form of the `mov` instruction; this adds another page table access first (to translate the array virtual address to the correct physical one) and then the array access itself.

The entire process, for the first five loop iterations, is depicted in Figure 18.7 (page 10). The bottom most graph shows the instruction memory references on the y-axis in black (with virtual addresses on the left, and the actual physical addresses on the right); the middle graph shows array accesses in dark gray (again with virtual on left and physical on right); finally, the topmost graph shows page table memory accesses in light gray (just physical, as the page table in this example resides in physical memory). The x-axis, for the entire trace, shows memory accesses across the first five iterations of the loop; there are 10 memory accesses per loop, which includes four instruction fetches, one explicit update of memory, and five page table accesses to translate those four fetches and one explicit update.

See if you can make sense of the patterns that show up in this visualization. In particular, what will change as the loop continues to run beyond these first five iterations? Which new memory locations will be accessed? Can you figure it out?

This has just been the simplest of examples (only a few lines of C code), and yet you might already be able to sense the complexity of understanding the actual memory behavior of real applications. Don't worry: it definitely gets worse, because the mechanisms we are about to introduce only complicate this already complex machinery. Sorry[3]!

18.6 Summary

We have introduced the concept of **paging** as a solution to our challenge of virtualizing memory. Paging has many advantages over previous approaches (such as segmentation). First, it does not lead to external fragmentation, as paging (by design) divides memory into fixed-sized units. Second, it is quite flexible, enabling the sparse use of virtual address spaces.

However, implementing paging support without care will lead to a slower machine (with many extra memory accesses to access the page table) as well as memory waste (with memory filled with page tables instead of useful application data). We'll thus have to think a little harder to come up with a paging system that not only works, but works well. The next two chapters, fortunately, will show us how to do so.

[3] We're not really sorry. But, we are sorry about not being sorry, if that makes sense.

References

[KE+62] "One-level Storage System"
T. Kilburn, and D.B.G. Edwards and M.J. Lanigan and F.H. Sumner
IRE Trans. EC-11, 2 (1962), pp. 223-235
(Reprinted in Bell and Newell, "Computer Structures: Readings and Examples" McGraw-Hill, New York, 1971).
The Atlas pioneered the idea of dividing memory into fixed-sized pages and in many senses was an early form of the memory-management ideas we see in modern computer systems.

[I09] "Intel 64 and IA-32 Architectures Software Developer's Manuals"
Intel, 2009
Available: http://www.intel.com/products/processor/manuals
In particular, pay attention to "Volume 3A: System Programming Guide Part 1" and "Volume 3B: System Programming Guide Part 2"

[L78] "The Manchester Mark I and atlas: a historical perspective"
S. H. Lavington
Communications of the ACM archive
Volume 21, Issue 1 (January 1978), pp. 4-12
Special issue on computer architecture
This paper is a great retrospective of some of the history of the development of some important computer systems. As we sometimes forget in the US, many of these new ideas came from overseas.

Homework

In this homework, you will use a simple program, which is known as `paging-linear-translate.py`, to see if you understand how simple virtual-to-physical address translation works with linear page tables. See the README for details.

Questions

1. Before doing any translations, let's use the simulator to study how linear page tables change size given different parameters. Compute the size of linear page tables as different parameters change. Some suggested inputs are below; by using the `-v flag`, you can see how many page-table entries are filled.
   First, to understand how linear page table size changes as the address space grows:

   ```
   paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
   paging-linear-translate.py -P 1k -a 2m -p 512m -v -n 0
   paging-linear-translate.py -P 1k -a 4m -p 512m -v -n 0
   ```

   Then, to understand how linear page table size changes as page size grows:

   ```
   paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
   paging-linear-translate.py -P 2k -a 1m -p 512m -v -n 0
   paging-linear-translate.py -P 4k -a 1m -p 512m -v -n 0
   ```

   Before running any of these, try to think about the expected trends. How should page-table size change as the address space grows? As the page size grows? Why shouldn't we just use really big pages in general?
2. Now let's do some translations. Start with some small examples, and change the number of pages that are allocated to the address space with the `-u flag`. For example:

   ```
   paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 0
   paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 25
   paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 50
   paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 75
   paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 100
   ```

   What happens as you increase the percentage of pages that are allocated in each address space?
3. Now let's try some different random seeds, and some different (and sometimes quite crazy) address-space parameters, for variety:

   ```
   paging-linear-translate.py -P 8  -a 32   -p 1024 -v -s 1
   paging-linear-translate.py -P 8k -a 32k  -p 1m   -v -s 2
   paging-linear-translate.py -P 1m -a 256m -p 512m -v -s 3
   ```

   Which of these parameter combinations are unrealistic? Why?

4. Use the program to try out some other problems. Can you find the limits of where the program doesn't work anymore? For example, what happens if the address-space size is *bigger* than physical memory?

19

Paging: Faster Translations (TLBs)

Using paging as the core mechanism to support virtual memory can lead to high performance overheads. By chopping the address space into small, fixed-sized units (i.e., pages), paging requires a large amount of mapping information. Because that mapping information is generally stored in physical memory, paging logically requires an extra memory lookup for each virtual address generated by the program. Going to memory for translation information before every instruction fetch or explicit load or store is prohibitively slow. And thus our problem:

THE CRUX:
HOW TO SPEED UP ADDRESS TRANSLATION

How can we speed up address translation, and generally avoid the extra memory reference that paging seems to require? What hardware support is required? What OS involvement is needed?

When we want to make things fast, the OS usually needs some help. And help often comes from the OS's old friend: the hardware. To speed address translation, we are going to add what is called (for historical reasons [CP78]) a **translation-lookaside buffer**, or **TLB** [C68, C95]. A TLB is part of the chip's **memory-management unit** (**MMU**), and is simply a hardware **cache** of popular virtual-to-physical address translations; thus, a better name would be an **address-translation cache**. Upon each virtual memory reference, the hardware first checks the TLB to see if the desired translation is held therein; if so, the translation is performed (quickly) *without* having to consult the page table (which has all translations). Because of their tremendous performance impact, TLBs in a real sense make virtual memory possible [C95].

19.1 TLB Basic Algorithm

Figure 19.1 shows a rough sketch of how hardware might handle a virtual address translation, assuming a simple **linear page table** (i.e., the page table is an array) and a **hardware-managed TLB** (i.e., the hardware handles much of the responsibility of page table accesses; we'll explain more about this below).

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)   // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset   = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10 else                  // TLB Miss
11     PTEAddr = PTBR + (VPN * sizeof(PTE))
12     PTE = AccessMemory(PTEAddr)
13     if (PTE.Valid == False)
14         RaiseException(SEGMENTATION_FAULT)
15     else if (CanAccess(PTE.ProtectBits) == False)
16         RaiseException(PROTECTION_FAULT)
17     else
18         TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19         RetryInstruction()
```

Figure 19.1: **TLB Control Flow Algorithm**

The algorithm the hardware follows works like this: first, extract the virtual page number (VPN) from the virtual address (Line 1 in Figure 19.1), and check if the TLB holds the translation for this VPN (Line 2). If it does, we have a **TLB hit**, which means the TLB holds the translation. Success! We can now extract the page frame number (PFN) from the relevant TLB entry, concatenate that onto the offset from the original virtual address, and form the desired physical address (PA), and access memory (Lines 5–7), assuming protection checks do not fail (Line 4).

If the CPU does not find the translation in the TLB (a **TLB miss**), we have some more work to do. In this example, the hardware accesses the page table to find the translation (Lines 11–12), and, assuming that the virtual memory reference generated by the process is valid and accessible (Lines 13, 15), updates the TLB with the translation (Line 18). These set of actions are costly, primarily because of the extra memory reference needed to access the page table (Line 12). Finally, once the TLB is updated, the hardware retries the instruction; this time, the translation is found in the TLB, and the memory reference is processed quickly.

The TLB, like all caches, is built on the premise that in the common case, translations are found in the cache (i.e., are hits). If so, little overhead is added, as the TLB is found near the processing core and is designed to be quite fast. When a miss occurs, the high cost of paging is incurred; the page table must be accessed to find the translation, and an extra memory reference (or more, with more complex page tables) results. If this happens often, the program will likely run noticeably more slowly; memory accesses, relative to most CPU instructions, are quite costly, and TLB misses lead to more memory accesses. Thus, it is our hope to avoid TLB misses as much as we can.

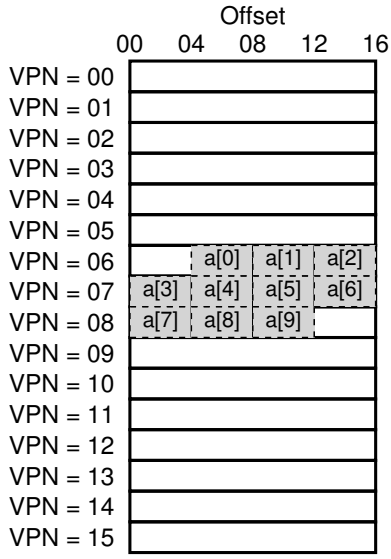


Figure 19.2: **Example: An Array In A Tiny Address Space**

19.2 Example: Accessing An Array

To make clear the operation of a TLB, let's examine a simple virtual address trace and see how a TLB can improve its performance. In this example, let's assume we have an array of 10 4-byte integers in memory, starting at virtual address 100. Assume further that we have a small 8-bit virtual address space, with 16-byte pages; thus, a virtual address breaks down into a 4-bit VPN (there are 16 virtual pages) and a 4-bit offset (there are 16 bytes on each of those pages).

Figure 19.2 shows the array laid out on the 16 16-byte pages of the system. As you can see, the array's first entry (`a[0]`) begins on (VPN=06, offset=04); only three 4-byte integers fit onto that page. The array continues onto the next page (VPN=07), where the next four entries (`a[3]` ... `a[6]`) are found. Finally, the last three entries of the 10-entry array (`a[7]` ... `a[9]`) are located on the next page of the address space (VPN=08).

Now let's consider a simple loop that accesses each array element, something that would look like this in C:

```
int sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i];
}
```

For the sake of simplicity, we will pretend that the only memory accesses the loop generates are to the array (ignoring the variables `i` and `sum`, as well as the instructions themselves). When the first array element (`a[0]`) is accessed, the CPU will see a load to virtual address 100. The hardware extracts the VPN from this (VPN=06), and uses that to check the TLB for a valid translation. Assuming this is the first time the program accesses the array, the result will be a TLB miss.

The next access is to `a[1]`, and there is some good news here: a TLB hit! Because the second element of the array is packed next to the first, it lives on the same page; because we've already accessed this page when accessing the first element of the array, the translation is already loaded into the TLB. And hence the reason for our success. Access to `a[2]` encounters similar success (another hit), because it too lives on the same page as `a[0]` and `a[1]`.

Unfortunately, when the program accesses `a[3]`, we encounter another TLB miss. However, once again, the next entries (`a[4]` ... `a[6]`) will hit in the TLB, as they all reside on the same page in memory.

Finally, access to `a[7]` causes one last TLB miss. The hardware once again consults the page table to figure out the location of this virtual page in physical memory, and updates the TLB accordingly. The final two accesses (`a[8]` and `a[9]`) receive the benefits of this TLB update; when the hardware looks in the TLB for their translations, two more hits result.

Let us summarize TLB activity during our ten accesses to the array: **miss**, hit, hit, **miss**, hit, hit, hit, **miss**, hit, hit. Thus, our TLB **hit rate**, which is the number of hits divided by the total number of accesses, is 70%. Although this is not too high (indeed, we desire hit rates that approach 100%), it is non-zero, which may be a surprise. Even though this is the first time the program accesses the array, the TLB improves performance due to **spatial locality**. The elements of the array are packed tightly into pages (i.e., they are close to one another in **space**), and thus only the first access to an element on a page yields a TLB miss.

Also note the role that page size plays in this example. If the page size had simply been twice as big (32 bytes, not 16), the array access would suffer even fewer misses. As typical page sizes are more like 4KB, these types of dense, array-based accesses achieve excellent TLB performance, encountering only a single miss per page of accesses.

One last point about TLB performance: if the program, soon after this loop completes, accesses the array again, we'd likely see an even better result, assuming that we have a big enough TLB to cache the needed translations: hit, hit, hit, hit, hit, hit, hit, hit, hit, hit. In this case, the TLB hit rate would be high because of **temporal locality**, i.e., the quick re-referencing of memory items in **time**. Like any cache, TLBs rely upon both spatial and temporal locality for success, which are program properties. If the program of interest exhibits such locality (and many programs do), the TLB hit rate will likely be high.

TIP: USE CACHING WHEN POSSIBLE

Caching is one of the most fundamental performance techniques in computer systems, one that is used again and again to make the "common-case fast" [HP06]. The idea behind hardware caches is to take advantage of **locality** in instruction and data references. There are usually two types of locality: **temporal locality** and **spatial locality**. With temporal locality, the idea is that an instruction or data item that has been recently accessed will likely be re-accessed soon in the future. Think of loop variables or instructions in a loop; they are accessed repeatedly over time. With spatial locality, the idea is that if a program accesses memory at address x, it will likely soon access memory near x. Imagine here streaming through an array of some kind, accessing one element and then the next. Of course, these properties depend on the exact nature of the program, and thus are not hard-and-fast laws but more like rules of thumb.

Hardware caches, whether for instructions, data, or address translations (as in our TLB) take advantage of locality by keeping copies of memory in small, fast on-chip memory. Instead of having to go to a (slow) memory to satisfy a request, the processor can first check if a nearby copy exists in a cache; if it does, the processor can access it quickly (i.e., in a few CPU cycles) and avoid spending the costly time it takes to access memory (many nanoseconds).

You might be wondering: if caches (like the TLB) are so great, why don't we just make bigger caches and keep all of our data in them? Unfortunately, this is where we run into more fundamental laws like those of physics. If you want a fast cache, it has to be small, as issues like the speed-of-light and other physical constraints become relevant. Any large cache by definition is slow, and thus defeats the purpose. Thus, we are stuck with small, fast caches; the question that remains is how to best use them to improve performance.

19.3 Who Handles The TLB Miss?

One question that we must answer: who handles a TLB miss? Two answers are possible: the hardware, or the software (OS). In the olden days, the hardware had complex instruction sets (sometimes called **CISC**, for complex-instruction set computers) and the people who built the hardware didn't much trust those sneaky OS people. Thus, the hardware would handle the TLB miss entirely. To do this, the hardware has to know exactly *where* the page tables are located in memory (via a **page-table base register**, used in Line 11 in Figure 19.1), as well as their *exact format*; on a miss, the hardware would "walk" the page table, find the correct page-table entry and extract the desired translation, update the TLB with the translation, and retry the instruction. An example of an "older" architecture that has **hardware-managed TLBs** is the Intel x86 architecture, which uses a fixed **multi-level page table** (see the next chapter for details); the current page table is pointed to by the CR3 register [I09].

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)   // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset   = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10 else                  // TLB Miss
11     RaiseException(TLB_MISS)
```

Figure 19.3: **TLB Control Flow Algorithm (OS Handled)**

More modern architectures (e.g., MIPS R10k [H93] or Sun's SPARC v9 [WG00], both **RISC** or reduced-instruction set computers) have what is known as a **software-managed TLB**. On a TLB miss, the hardware simply raises an exception (line 11 in Figure 19.3), which pauses the current instruction stream, raises the privilege level to kernel mode, and jumps to a **trap handler**. As you might guess, this trap handler is code within the OS that is written with the express purpose of handling TLB misses. When run, the code will lookup the translation in the page table, use special "privileged" instructions to update the TLB, and return from the trap; at this point, the hardware retries the instruction (resulting in a TLB hit).

Let's discuss a couple of important details. First, the return-from-trap instruction needs to be a little different than the return-from-trap we saw before when servicing a system call. In the latter case, the return-from-trap should resume execution at the instruction *after* the trap into the OS, just as a return from a procedure call returns to the instruction immediately following the call into the procedure. In the former case, when returning from a TLB miss-handling trap, the hardware must resume execution at the instruction that *caused* the trap; this retry thus lets the instruction run again, this time resulting in a TLB hit. Thus, depending on how a trap or exception was caused, the hardware must save a different PC when trapping into the OS, in order to resume properly when the time to do so arrives.

Second, when running the TLB miss-handling code, the OS needs to be extra careful not to cause an infinite chain of TLB misses to occur. Many solutions exist; for example, you could keep TLB miss handlers in physical memory (where they are **unmapped** and not subject to address translation), or reserve some entries in the TLB for permanently-valid translations and use some of those permanent translation slots for the handler code itself; these **wired** translations always hit in the TLB.

The primary advantage of the software-managed approach is *flexibility*: the OS can use any data structure it wants to implement the page table, without necessitating hardware change. Another advantage is *simplicity*; as you can see in the TLB control flow (line 11 in Figure 19.3, in contrast to lines 11–19 in Figure 19.1), the hardware doesn't have to do much on a miss; it raises an exception, and the OS TLB miss handler does the rest.

ASIDE: **RISC VS. CISC**

In the 1980's, a great battle took place in the computer architecture community. On one side was the **CISC** camp, which stood for **Complex Instruction Set Computing**; on the other side was **RISC**, for **Reduced Instruction Set Computing** [PS81]. The RISC side was spear-headed by David Patterson at Berkeley and John Hennessy at Stanford (who are also co-authors of some famous books [HP06]), although later John Cocke was recognized with a Turing award for his earliest work on RISC [CM00].

CISC instruction sets tend to have a lot of instructions in them, and each instruction is relatively powerful. For example, you might see a string copy, which takes two pointers and a length and copies bytes from source to destination. The idea behind CISC was that instructions should be high-level primitives, to make the assembly language itself easier to use, and to make code more compact.

RISC instruction sets are exactly the opposite. A key observation behind RISC is that instruction sets are really compiler targets, and all compilers really want are a few simple primitives that they can use to generate high-performance code. Thus, RISC proponents argued, let's rip out as much from the hardware as possible (especially the microcode), and make what's left simple, uniform, and fast.

In the early days, RISC chips made a huge impact, as they were noticeably faster [BC91]; many papers were written; a few companies were formed (e.g., MIPS and Sun). However, as time progressed, CISC manufacturers such as Intel incorporated many RISC techniques into the core of their processors, for example by adding early pipeline stages that transformed complex instructions into micro-instructions which could then be processed in a RISC-like manner. These innovations, plus a growing number of transistors on each chip, allowed CISC to remain competitive. The end result is that the debate died down, and today both types of processors can be made to run fast.

19.4 TLB Contents: What's In There?

Let's look at the contents of the hardware TLB in more detail. A typical TLB might have 32, 64, or 128 entries and be what is called **fully associative**. Basically, this just means that any given translation can be anywhere in the TLB, and that the hardware will search the entire TLB in parallel to find the desired translation. A TLB entry might look like this:

VPN | PFN | other bits

Note that both the VPN and PFN are present in each entry, as a translation could end up in any of these locations (in hardware terms, the TLB is known as a **fully-associative** cache). The hardware searches the entries in parallel to see if there is a match.

ASIDE: **TLB VALID BIT $\neq$ PAGE TABLE VALID BIT**

A common mistake is to confuse the valid bits found in a TLB with those found in a page table. In a page table, when a page-table entry (PTE) is marked invalid, it means that the page has not been allocated by the process, and should not be accessed by a correctly-working program. The usual response when an invalid page is accessed is to trap to the OS, which will respond by killing the process.

A TLB valid bit, in contrast, simply refers to whether a TLB entry has a valid translation within it. When a system boots, for example, a common initial state for each TLB entry is to be set to invalid, because no address translations are yet cached there. Once virtual memory is enabled, and once programs start running and accessing their virtual address spaces, the TLB is slowly populated, and thus valid entries soon fill the TLB.

The TLB valid bit is quite useful when performing a context switch too, as we'll discuss further below. By setting all TLB entries to invalid, the system can ensure that the about-to-be-run process does not accidentally use a virtual-to-physical translation from a previous process.

More interesting are the "other bits". For example, the TLB commonly has a **valid** bit, which says whether the entry has a valid translation or not. Also common are **protection** bits, which determine how a page can be accessed (as in the page table). For example, code pages might be marked *read and execute*, whereas heap pages might be marked *read and write*. There may also be a few other fields, including an **address-space identifier**, a **dirty bit**, and so forth; see below for more information.

19.5 TLB Issue: Context Switches

With TLBs, some new issues arise when switching between processes (and hence address spaces). Specifically, the TLB contains virtual-to-physical translations that are only valid for the currently running process; these translations are not meaningful for other processes. As a result, when switching from one process to another, the hardware or OS (or both) must be careful to ensure that the about-to-be-run process does not accidentally use translations from some previously run process.

To understand this situation better, let's look at an example. When one process (P1) is running, it assumes the TLB might be caching translations that are valid for it, i.e., that come from P1's page table. Assume, for this example, that the 10th virtual page of P1 is mapped to physical frame 100.

In this example, assume another process (P2) exists, and the OS soon might decide to perform a context switch and run it. Assume here that the 10th virtual page of P2 is mapped to physical frame 170. If entries for both processes were in the TLB, the contents of the TLB would be:

VPN	PFN	valid	prot
10	100	1	rwx
—	—	0	—
10	170	1	rwx
—	—	0	—

In the TLB above, we clearly have a problem: VPN 10 translates to either PFN 100 (P1) or PFN 170 (P2), but the hardware can't distinguish which entry is meant for which process. Thus, we need to do some more work in order for the TLB to correctly and efficiently support virtualization across multiple processes. And thus, a crux:

THE CRUX:
HOW TO MANAGE TLB CONTENTS ON A CONTEXT SWITCH
When context-switching between processes, the translations in the TLB for the last process are not meaningful to the about-to-be-run process. What should the hardware or OS do in order to solve this problem?

There are a number of possible solutions to this problem. One approach is to simply **flush** the TLB on context switches, thus emptying it before running the next process. On a software-based system, this can be accomplished with an explicit (and privileged) hardware instruction; with a hardware-managed TLB, the flush could be enacted when the page-table base register is changed (note the OS must change the PTBR on a context switch anyhow). In either case, the flush operation simply sets all valid bits to 0, essentially clearing the contents of the TLB.

By flushing the TLB on each context switch, we now have a working solution, as a process will never accidentally encounter the wrong translations in the TLB. However, there is a cost: each time a process runs, it must incur TLB misses as it touches its data and code pages. If the OS switches between processes frequently, this cost may be high.

To reduce this overhead, some systems add hardware support to enable sharing of the TLB across context switches. In particular, some hardware systems provide an **address space identifier** (**ASID**) field in the TLB. You can think of the ASID as a **process identifier** (**PID**), but usually it has fewer bits (e.g., 8 bits for the ASID versus 32 bits for a PID).

If we take our example TLB from above and add ASIDs, it is clear processes can readily share the TLB: only the ASID field is needed to differentiate otherwise identical translations. Here is a depiction of a TLB with the added ASID field:

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

Thus, with address-space identifiers, the TLB can hold translations from different processes at the same time without any confusion. Of course, the hardware also needs to know which process is currently running in order to perform translations, and thus the OS must, on a context switch, set some privileged register to the ASID of the current process.

As an aside, you may also have thought of another case where two entries of the TLB are remarkably similar. In this example, there are two entries for two different processes with two different VPNs that point to the *same* physical page:

VPN	PFN	valid	prot	ASID
10	101	1	r-x	1
—	—	0	—	—
50	101	1	r-x	2
—	—	0	—	—

This situation might arise, for example, when two processes *share* a page (a code page, for example). In the example above, Process 1 is sharing physical page 101 with Process 2; P1 maps this page into the 10th page of its address space, whereas P2 maps it to the 50th page of its address space. Sharing of code pages (in binaries, or shared libraries) is useful as it reduces the number of physical pages in use, thus reducing memory overheads.

19.6 Issue: Replacement Policy

As with any cache, and thus also with the TLB, one more issue that we must consider is **cache replacement**. Specifically, when we are installing a new entry in the TLB, we have to **replace** an old one, and thus the question: which one to replace?

THE CRUX: HOW TO DESIGN TLB REPLACEMENT POLICY

Which TLB entry should be replaced when we add a new TLB entry? The goal, of course, being to minimize the **miss rate** (or increase **hit rate**) and thus improve performance.

We will study such policies in some detail when we tackle the problem of swapping pages to disk; here we'll just highlight a few typical policies. One common approach is to evict the **least-recently-used** or **LRU** entry. LRU tries to take advantage of locality in the memory-reference stream, assuming it is likely that an entry that has not recently been used is a good candidate for eviction. Another typical approach is to use a **random** policy, which evicts a TLB mapping at random. Such a policy is useful due to its simplicity and ability to avoid corner-case behaviors; for example, a "reasonable" policy such as LRU behaves quite unreasonably when a program loops over $n + 1$ pages with a TLB of size n; in this case, LRU misses upon every access, whereas random does much better.

Figure 19.4: **A MIPS TLB Entry**

19.7 A Real TLB Entry

Finally, let's briefly look at a real TLB. This example is from the MIPS R4000 [H93], a modern system that uses software-managed TLBs; a slightly simplified MIPS TLB entry can be seen in Figure 19.4.

The MIPS R4000 supports a 32-bit address space with 4KB pages. Thus, we would expect a 20-bit VPN and 12-bit offset in our typical virtual address. However, as you can see in the TLB, there are only 19 bits for the VPN; as it turns out, user addresses will only come from half the address space (the rest reserved for the kernel) and hence only 19 bits of VPN are needed. The VPN translates to up to a 24-bit physical frame number (PFN), and hence can support systems with up to 64GB of (physical) main memory (2^{24} 4KB pages).

There are a few other interesting bits in the MIPS TLB. We see a *global* bit (G), which is used for pages that are globally-shared among processes. Thus, if the global bit is set, the ASID is ignored. We also see the 8-bit *ASID*, which the OS can use to distinguish between address spaces (as described above). One question for you: what should the OS do if there are more than 256 (2^8) processes running at a time? Finally, we see 3 *Coherence* (C) bits, which determine how a page is cached by the hardware (a bit beyond the scope of these notes); a *dirty* bit which is marked when the page has been written to (we'll see the use of this later); a *valid* bit which tells the hardware if there is a valid translation present in the entry. There is also a *page mask* field (not shown), which supports multiple page sizes; we'll see later why having larger pages might be useful. Finally, some of the 64 bits are unused (shaded gray in the diagram).

MIPS TLBs usually have 32 or 64 of these entries, most of which are used by user processes as they run. However, a few are reserved for the OS. A *wired* register can be set by the OS to tell the hardware how many slots of the TLB to reserve for the OS; the OS uses these reserved mappings for code and data that it wants to access during critical times, where a TLB miss would be problematic (e.g., in the TLB miss handler).

Because the MIPS TLB is software managed, there needs to be instructions to update the TLB. The MIPS provides four such instructions: `TLBP`, which probes the TLB to see if a particular translation is in there; `TLBR`, which reads the contents of a TLB entry into registers; `TLBWI`, which replaces a specific TLB entry; and `TLBWR`, which replaces a random TLB entry. The OS uses these instructions to manage the TLB's contents. It is of course critical that these instructions are **privileged**; imagine what a user process could do if it could modify the contents of the TLB (hint: just about anything, including take over the machine, run its own malicious "OS", or even make the Sun disappear).

TIP: RAM ISN'T ALWAYS RAM (CULLER'S LAW)
The term **random-access memory**, or **RAM**, implies that you can access any part of RAM just as quickly as another. While it is generally good to think of RAM in this way, because of hardware/OS features such as the TLB, accessing a particular page of memory may be costly, particularly if that page isn't currently mapped by your TLB. Thus, it is always good to remember the implementation tip: **RAM isn't always RAM**. Sometimes randomly accessing your address space, particular if the number of pages accessed exceeds the TLB coverage, can lead to severe performance penalties. Because one of our advisors, David Culler, used to always point to the TLB as the source of many performance problems, we name this law in his honor: **Culler's Law**.

19.8 Summary

We have seen how hardware can help us make address translation faster. By providing a small, dedicated on-chip TLB as an address-translation cache, most memory references will hopefully be handled *without* having to access the page table in main memory. Thus, in the common case, the performance of the program will be almost as if memory isn't being virtualized at all, an excellent achievement for an operating system, and certainly essential to the use of paging in modern systems.

However, TLBs do not make the world rosy for every program that exists. In particular, if the number of pages a program accesses in a short period of time exceeds the number of pages that fit into the TLB, the program will generate a large number of TLB misses, and thus run quite a bit more slowly. We refer to this phenomenon as exceeding the **TLB coverage**, and it can be quite a problem for certain programs. One solution, as we'll discuss in the next chapter, is to include support for larger page sizes; by mapping key data structures into regions of the program's address space that are mapped by larger pages, the effective coverage of the TLB can be increased. Support for large pages is often exploited by programs such as a **database management system** (a **DBMS**), which have certain data structures that are both large and randomly-accessed.

One other TLB issue worth mentioning: TLB access can easily become a bottleneck in the CPU pipeline, in particular with what is called a **physically-indexed cache**. With such a cache, address translation has to take place *before* the cache is accessed, which can slow things down quite a bit. Because of this potential problem, people have looked into all sorts of clever ways to access caches with *virtual* addresses, thus avoiding the expensive step of translation in the case of a cache hit. Such a **virtually-indexed cache** solves some performance problems, but introduces new issues into hardware design as well. See Wiggins's fine survey for more details [W03].

References

[BC91] "Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization"
D. Bhandarkar and Douglas W. Clark
Communications of the ACM, September 1991
A great and fair comparison between RISC and CISC. The bottom line: on similar hardware, RISC was about a factor of three better in performance.

[CM00] "The evolution of RISC technology at IBM"
John Cocke and V. Markstein
IBM Journal of Research and Development, 44:1/2
A summary of the ideas and work behind the IBM 801, which many consider the first true RISC microprocessor.

[C95] "The Core of the Black Canyon Computer Corporation"
John Couleur
IEEE Annals of History of Computing, 17:4, 1995
In this fascinating historical note, Couleur talks about how he invented the TLB in 1964 while working for GE, and the fortuitous collaboration that thus ensued with the Project MAC folks at MIT.

[CG68] "Shared-access Data Processing System"
John F. Couleur and Edward L. Glaser
Patent 3412382, November 1968
The patent that contains the idea for an associative memory to store address translations. The idea, according to Couleur, came in 1964.

[CP78] "The architecture of the IBM System/370"
R.P. Case and A. Padegs
Communications of the ACM. 21:1, 73-96, January 1978
*Perhaps the first paper to use the term **translation lookaside buffer**. The name arises from the historical name for a cache, which was a **lookaside buffer** as called by those developing the Atlas system at the University of Manchester; a cache of address translations thus became a **translation lookaside buffer**. Even though the term lookaside buffer fell out of favor, TLB seems to have stuck, for whatever reason.*

[H93] "MIPS R4000 Microprocessor User's Manual".
Joe Heinrich, Prentice-Hall, June 1993
Available: http://cag.csail.mit.edu/raw/documents/R4400_Uman_book_Ed2.pdf

[HP06] "Computer Architecture: A Quantitative Approach"
John Hennessy and David Patterson
Morgan-Kaufmann, 2006
A great book about computer architecture. We have a particular attachment to the classic first edition.

[I09] "Intel 64 and IA-32 Architectures Software Developer's Manuals"
Intel, 2009
Available: http://www.intel.com/products/processor/manuals
In particular, pay attention to "Volume 3A: System Programming Guide Part 1" and "Volume 3B: System Programming Guide Part 2"

[PS81] "RISC-I: A Reduced Instruction Set VLSI Computer"
D.A. Patterson and C.H. Sequin
ISCA '81, Minneapolis, May 1981
The paper that introduced the term RISC, and started the avalanche of research into simplifying computer chips for performance.

[SB92] "CPU Performance Evaluation and Execution Time Prediction
Using Narrow Spectrum Benchmarking"
Rafael H. Saavedra-Barrera
EECS Department, University of California, Berkeley
Technical Report No. UCB/CSD-92-684, February 1992
www.eecs.berkeley.edu/Pubs/TechRpts/1992/CSD-92-684.pdf
A great dissertation about how to predict execution time of applications by breaking them down into constituent pieces and knowing the cost of each piece. Probably the most interesting part that comes out of this work is the tool to measure details of the cache hierarchy (described in Chapter 5). Make sure to check out the wonderful diagrams therein.

[W03] "A Survey on the Interaction Between Caching, Translation and Protection"
Adam Wiggins
University of New South Wales TR UNSW-CSE-TR-0321, August, 2003
An excellent survey of how TLBs interact with other parts of the CPU pipeline, namely hardware caches.

[WG00] "The SPARC Architecture Manual: Version 9"
David L. Weaver and Tom Germond, September 2000
SPARC International, San Jose, California
Available: http://www.sparc.org/standards/SPARCV9.pdf

Homework (Measurement)

In this homework, you are to measure the size and cost of accessing a TLB. The idea is based on work by Saavedra-Barrera [SB92], who developed a simple but beautiful method to measure numerous aspects of cache hierarchies, all with a very simple user-level program. Read his work for more details.

The basic idea is to access some number of pages within a large data structure (e.g., an array) and to time those accesses. For example, let's say the TLB size of a machine happens to be 4 (which would be very small, but useful for the purposes of this discussion). If you write a program that touches 4 or fewer pages, each access should be a TLB hit, and thus relatively fast. However, once you touch 5 pages or more, repeatedly in a loop, each access will suddenly jump in cost, to that of a TLB miss.

The basic code to loop through an array once should look like this:

```
int jump = PAGESIZE / sizeof(int);
for (i = 0; i < NUMPAGES * jump; i += jump) {
    a[i] += 1;
}
```

In this loop, one integer per page of the array `a` is updated, up to the number of pages specified by `NUMPAGES`. By timing such a loop repeatedly (say, a few hundred million times in another loop around this one, or however many loops are needed to run for a few seconds), you can time how long each access takes (on average). By looking for jumps in cost as `NUMPAGES` increases, you can roughly determine how big the first-level TLB is, determine whether a second-level TLB exists (and how big it is if it does), and in general get a good sense of how TLB hits and misses can affect performance.

Figure 19.5 (page 16) shows the average time per access as the number of pages accessed in the loop is increased. As you can see in the graph, when just a few pages are accessed (8 or fewer), the average access time is roughly 5 nanoseconds. When 16 or more pages are accessed, there is a sudden jump to about 20 nanoseconds per access. A final jump in cost occurs at around 1024 pages, at which point each access takes around 70 nanoseconds. From this data, we can conclude that there is a two-level TLB hierarchy; the first is quite small (probably holding between 8 and 16 entries); the second is larger but slower (holding roughly 512 entries). The overall difference between hits in the first-level TLB and misses is quite large, roughly a factor of fourteen. TLB performance matters!

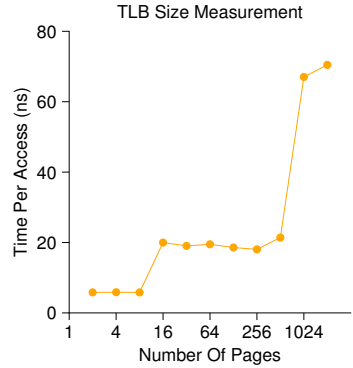


Figure 19.5: **Discovering TLB Sizes and Miss Costs**

Questions

1. For timing, you'll need to use a timer such as that made available by `gettimeofday()`. How precise is such a timer? How long does an operation have to take in order for you to time it precisely? (this will help determine how many times, in a loop, you'll have to repeat a page access in order to time it successfully)
2. Write the program, called `tlb.c`, that can roughly measure the cost of accessing each page. Inputs to the program should be: the number of pages to touch and the number of trials.
3. Now write a script in your favorite scripting language (csh, python, etc.) to run this program, while varying the number of pages accessed from 1 up to a few thousand, perhaps incrementing by a factor of two per iteration. Run the script on different machines and gather some data. How many trials are needed to get reliable measurements?
4. Next, graph the results, making a graph that looks similar to the one above. Use a good tool like `ploticus`. Visualization usually makes the data much easier to digest; why do you think that is?
5. One thing to watch out for is compiler optimzation. Compilers do all sorts of clever things, including removing loops which increment values that no other part of the program subsequently uses. How can you ensure the compiler does not remove the main loop above from your TLB size estimator?
6. Another thing to watch out for is the fact that most systems today ship with multiple CPUs, and each CPU, of course, has its own TLB hierarchy. To really get good measurements, you have to run your

code on just one CPU, instead of letting the scheduler bounce it from one CPU to the next. How can you do that? (hint: look up "pinning a thread" on Google for some clues) What will happen if you don't do this, and the code moves from one CPU to the other?

7. Another issue that might arise relates to initialization. If you don't initialize the array `a` above before accessing it, the first time you access it will be very expensive, due to initial access costs such as demand zeroing. Will this affect your code and its timing? What can you do to counterbalance these potential costs?

20

Paging: Smaller Tables

We now tackle the second problem that paging introduces: page tables are too big and thus consume too much memory. Let's start out with a linear page table. As you might recall[1], linear page tables get pretty big. Assume again a 32-bit address space (2^{32} bytes), with 4KB (2^{12} byte) pages and a 4-byte page-table entry. An address space thus has roughly one million virtual pages in it ($\frac{2^{32}}{2^{12}}$); multiply by the page-table entry size and you see that our page table is 4MB in size. Recall also: we usually have one page table *for every process* in the system! With a hundred active processes (not uncommon on a modern system), we will be allocating hundreds of megabytes of memory just for page tables! As a result, we are in search of some techniques to reduce this heavy burden. There are a lot of them, so let's get going. But not before our crux:

CRUX: HOW TO MAKE PAGE TABLES SMALLER?
Simple array-based page tables (usually called linear page tables) are too big, taking up far too much memory on typical systems. How can we make page tables smaller? What are the key ideas? What inefficiencies arise as a result of these new data structures?

20.1 Simple Solution: Bigger Pages

We could reduce the size of the page table in one simple way: use bigger pages. Take our 32-bit address space again, but this time assume 16KB pages. We would thus have an 18-bit VPN plus a 14-bit offset. Assuming the same size for each PTE (4 bytes), we now have 2^{18} entries in our linear page table and thus a total size of 1MB per page table, a factor

[1]Or indeed, you might not; this paging thing is getting out of control, no? That said, always make sure you understand the *problem* you are solving before moving onto the solution; indeed, if you understand the problem, you can often derive the solution yourself. Here, the problem should be clear: simple linear (array-based) page tables are too big.

ASIDE: **MULTIPLE PAGE SIZES**
As an aside, do note that many architectures (e.g., MIPS, SPARC, x86-64) now support multiple page sizes. Usually, a small (4KB or 8KB) page size is used. However, if a "smart" application requests it, a single large page (e.g., of size 4MB) can be used for a specific portion of the address space, enabling such applications to place a frequently-used (and large) data structure in such a space while consuming only a single TLB entry. This type of large page usage is common in database management systems and other high-end commercial applications. The main reason for multiple page sizes is not to save page table space, however; it is to reduce pressure on the TLB, enabling a program to access more of its address space without suffering from too many TLB misses. However, as researchers have shown [N+02], using multiple page sizes makes the OS virtual memory manager notably more complex, and thus large pages are sometimes most easily used simply by exporting a new interface to applications to request large pages directly.

of four reduction in size of the page table (not surprisingly, the reduction exactly mirrors the factor of four increase in page size).

The major problem with this approach, however, is that big pages lead to waste *within* each page, a problem known as **internal fragmentation** (as the waste is **internal** to the unit of allocation). Applications thus end up allocating pages but only using little bits and pieces of each, and memory quickly fills up with these overly-large pages. Thus, most systems use relatively small page sizes in the common case: 4KB (as in x86) or 8KB (as in SPARCv9). Our problem will not be solved so simply, alas.

20.2 Hybrid Approach: Paging and Segments

Whenever you have two reasonable but different approaches to something in life, you should always examine the combination of the two to see if you can obtain the best of both worlds. We call such a combination a **hybrid**. For example, why eat just chocolate or plain peanut butter when you can instead combine the two in a lovely hybrid known as the Reese's Peanut Butter Cup [M28]?

Years ago, the creators of Multics (in particular Jack Dennis) chanced upon such an idea in the construction of the Multics virtual memory system [M07]. Specifically, Dennis had the idea of combining paging and segmentation in order to reduce the memory overhead of page tables. We can see why this might work by examining a typical linear page table in more detail. Assume we have an address space in which the used portions of the heap and stack are small. For the example, we use a tiny 16KB address space with 1KB pages (Figure 20.1); the page table for this address space is in Figure 20.2.

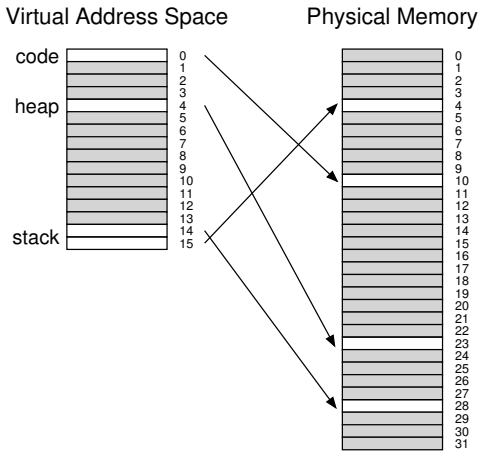


Figure 20.1: **A 16KB Address Space With 1KB Pages**

This example assumes the single code page (VPN 0) is mapped to physical page 10, the single heap page (VPN 4) to physical page 23, and the two stack pages at the other end of the address space (VPNs 14 and 15) are mapped to physical pages 28 and 4, respectively. As you can see from the picture, *most* of the page table is unused, full of **invalid** entries. What a waste! And this is for a tiny 16KB address space. Imagine the page table of a 32-bit address space and all the potential wasted space in there! Actually, don't imagine such a thing; it's far too gruesome.

PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
23	1	rw-	1	1
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
28	1	rw-	1	1
4	1	rw-	1	1

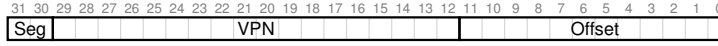
Figure 20.2: **A Page Table For 16KB Address Space**

Thus, our hybrid approach: instead of having a single page table for the entire address space of the process, why not have one per logical segment? In this example, we might thus have three page tables, one for the code, heap, and stack parts of the address space.

Now, remember with segmentation, we had a **base** register that told us where each segment lived in physical memory, and a **bound** or **limit** register that told us the size of said segment. In our hybrid, we still have those structures in the MMU; here, we use the base not to point to the segment itself but rather to hold the *physical address of the page table* of that segment. The bounds register is used to indicate the end of the page table (i.e., how many valid pages it has).

Let's do a simple example to clarify. Assume a 32-bit virtual address space with 4KB pages, and an address space split into four segments. We'll only use three segments for this example: one for code, one for heap, and one for stack.

To determine which segment an address refers to, we'll use the top two bits of the address space. Let's assume 00 is the unused segment, with 01 for code, 10 for the heap, and 11 for the stack. Thus, a virtual address looks like this:



In the hardware, assume that there are thus three base/bounds pairs, one each for code, heap, and stack. When a process is running, the base register for each of these segments contains the physical address of a linear page table for that segment; thus, each process in the system now has *three* page tables associated with it. On a context switch, these registers must be changed to reflect the location of the page tables of the newly-running process.

On a TLB miss (assuming a hardware-managed TLB, i.e., where the hardware is responsible for handling TLB misses), the hardware uses the segment bits (`SN`) to determine which base and bounds pair to use. The hardware then takes the physical address therein and combines it with the VPN as follows to form the address of the page table entry (PTE):

```
SN           = (VirtualAddress & SEG_MASK) >> SN_SHIFT
VPN          = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
```

This sequence should look familiar; it is virtually identical to what we saw before with linear page tables. The only difference, of course, is the use of one of three segment base registers instead of the single page table base register.

The critical difference in our hybrid scheme is the presence of a bounds register per segment; each bounds register holds the value of the maximum valid page in the segment. For example, if the code segment is using its first three pages (0, 1, and 2), the code segment page table will only have three entries allocated to it and the bounds register will be set

TIP: USE HYBRIDS

When you have two good and seemingly opposing ideas, you should always see if you can combine them into a **hybrid** that manages to achieve the best of both worlds. Hybrid corn species, for example, are known to be more robust than any naturally-occurring species. Of course, not all hybrids are a good idea; see the Zeedonk (or Zonkey), which is a cross of a Zebra and a Donkey. If you don't believe such a creature exists, look it up, and prepare to be amazed.

to 3; memory accesses beyond the end of the segment will generate an exception and likely lead to the termination of the process. In this manner, our hybrid approach realizes a significant memory savings compared to the linear page table; unallocated pages between the stack and the heap no longer take up space in a page table (just to mark them as not valid).

However, as you might notice, this approach is not without problems. First, it still requires us to use segmentation; as we discussed before, segmentation is not quite as flexible as we would like, as it assumes a certain usage pattern of the address space; if we have a large but sparsely-used heap, for example, we can still end up with a lot of page table waste. Second, this hybrid causes external fragmentation to arise again. While most of memory is managed in page-sized units, page tables now can be of arbitrary size (in multiples of PTEs). Thus, finding free space for them in memory is more complicated. For these reasons, people continued to look for better ways to implement smaller page tables.

20.3 Multi-level Page Tables

A different approach doesn't rely on segmentation but attacks the same problem: how to get rid of all those invalid regions in the page table instead of keeping them all in memory? We call this approach a **multi-level page table**, as it turns the linear page table into something like a tree. This approach is so effective that many modern systems employ it (e.g., x86 [BOH10]). We now describe this approach in detail.

The basic idea behind a multi-level page table is simple. First, chop up the page table into page-sized units; then, if an entire page of page-table entries (PTEs) is invalid, don't allocate that page of the page table at all. To track whether a page of the page table is valid (and if valid, where it is in memory), use a new structure, called the **page directory**. The page directory thus either can be used to tell you where a page of the page table is, or that the entire page of the page table contains no valid pages.

Figure 20.3 shows an example. On the left of the figure is the classic linear page table; even though most of the middle regions of the address space are not valid, we still require page-table space allocated for those regions (i.e., the middle two pages of the page table). On the right is a multi-level page table. The page directory marks just two pages of the

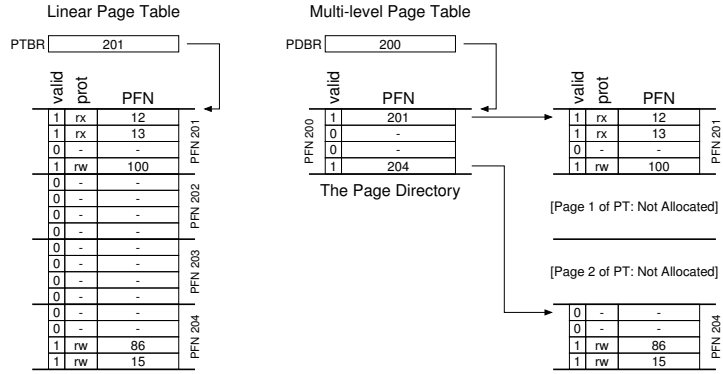


Figure 20.3: **Linear (Left) And Multi-Level (Right) Page Tables**

page table as valid (the first and last); thus, just those two pages of the page table reside in memory. And thus you can see one way to visualize what a multi-level table is doing: it just makes parts of the linear page table disappear (freeing those frames for other uses), and tracks which pages of the page table are allocated with the page directory.

The page directory, in a simple two-level table, contains one entry per page of the page table. It consists of a number of **page directory entries** (**PDE**). A PDE (minimally) has a **valid bit** and a **page frame number** (PFN), similar to a PTE. However, as hinted at above, the meaning of this valid bit is slightly different: if the PDE entry is valid, it means that at least one of the pages of the page table that the entry points to (via the PFN) is valid, i.e., in at least one PTE on that page pointed to by this PDE, the valid bit in that PTE is set to one. If the PDE entry is not valid (i.e., equal to zero), the rest of the PDE is not defined.

Multi-level page tables have some obvious advantages over approaches we've seen thus far. First, and perhaps most obviously, the multi-level table only allocates page-table space in proportion to the amount of address space you are using; thus it is generally compact and supports sparse address spaces.

Second, if carefully constructed, each portion of the page table fits neatly within a page, making it easier to manage memory; the OS can simply grab the next free page when it needs to allocate or grow a page table. Contrast this to a simple (non-paged) linear page table[2], which is just an array of PTEs indexed by VPN; with such a structure, the entire linear page table must reside contiguously in physical memory. For a large page table (say 4MB), finding such a large chunk of unused contiguous free physical memory can be quite a challenge. With a multi-level

[2]We are making some assumptions here, i.e., that all page tables reside in their entirety in physical memory (i.e., they are not swapped to disk); we'll soon relax this assumption.

TIP: UNDERSTAND TIME-SPACE TRADE-OFFS
When building a data structure, one should always consider **time-space trade-offs** in its construction. Usually, if you wish to make access to a particular data structure faster, you will have to pay a space-usage penalty for the structure.

structure, we add a **level of indirection** through use of the page directory, which points to pieces of the page table; that indirection allows us to place page-table pages wherever we would like in physical memory.

It should be noted that there is a cost to multi-level tables; on a TLB miss, two loads from memory will be required to get the right translation information from the page table (one for the page directory, and one for the PTE itself), in contrast to just one load with a linear page table. Thus, the multi-level table is a small example of a **time-space trade-off**. We wanted smaller tables (and got them), but not for free; although in the common case (TLB hit), performance is obviously identical, a TLB miss suffers from a higher cost with this smaller table.

Another obvious negative is *complexity*. Whether it is the hardware or OS handling the page-table lookup (on a TLB miss), doing so is undoubtedly more involved than a simple linear page-table lookup. Often we are willing to increase complexity in order to improve performance or reduce overheads; in the case of a multi-level table, we make page-table lookups more complicated in order to save valuable memory.

A Detailed Multi-Level Example

To understand the idea behind multi-level page tables better, let's do an example. Imagine a small address space of size 16KB, with 64-byte pages. Thus, we have a 14-bit virtual address space, with 8 bits for the VPN and 6 bits for the offset. A linear page table would have 2^8 (256) entries, even if only a small portion of the address space is in use. Figure 20.4 presents one example of such an address space.

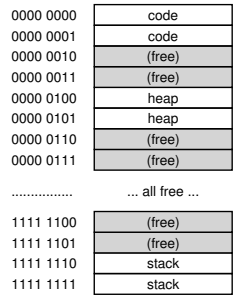


Figure 20.4: **A 16KB Address Space With 64-byte Pages**

TIP: BE WARY OF COMPLEXITY

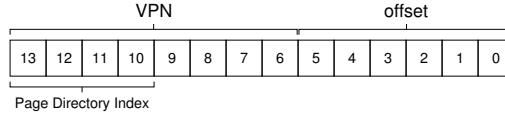
System designers should be wary of adding complexity into their system. What a good systems builder does is implement the least complex system that achieves the task at hand. For example, if disk space is abundant, you shouldn't design a file system that works hard to use as few bytes as possible; similarly, if processors are fast, it is better to write a clean and understandable module within the OS than perhaps the most CPU-optimized, hand-assembled code for the task at hand. Be wary of needless complexity, in prematurely-optimized code or other forms; such approaches make systems harder to understand, maintain, and debug. As Antoine de Saint-Exupery famously wrote: "Perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away." What he didn't write: "It's a lot easier to say something about perfection than to actually achieve it."

In this example, virtual pages 0 and 1 are for code, virtual pages 4 and 5 for the heap, and virtual pages 254 and 255 for the stack; the rest of the pages of the address space are unused.

To build a two-level page table for this address space, we start with our full linear page table and break it up into page-sized units. Recall our full table (in this example) has 256 entries; assume each PTE is 4 bytes in size. Thus, our page table is 1KB (256 $\times$ 4 bytes) in size. Given that we have 64-byte pages, the 1KB page table can be divided into 16 64-byte pages; each page can hold 16 PTEs.

What we need to understand now is how to take a VPN and use it to index first into the page directory and then into the page of the page table. Remember that each is an array of entries; thus, all we need to figure out is how to construct the index for each from pieces of the VPN.

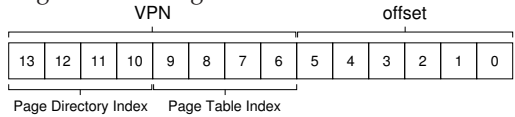
Let's first index into the page directory. Our page table in this example is small: 256 entries, spread across 16 pages. The page directory needs one entry per page of the page table; thus, it has 16 entries. As a result, we need four bits of the VPN to index into the directory; we use the top four bits of the VPN, as follows:



Once we extract the **page-directory index** (`PDIndex` for short) from the VPN, we can use it to find the address of the page-directory entry (PDE) with a simple calculation: `PDEAddr = PageDirBase + (PDIndex * sizeof(PDE))`. This results in our page directory, which we now examine to make further progress in our translation.

If the page-directory entry is marked invalid, we know that the access is invalid, and thus raise an exception. If, however, the PDE is valid,

we have more work to do. Specifically, we now have to fetch the page-table entry (PTE) from the page of the page table pointed to by this page-directory entry. To find this PTE, we have to index into the portion of the page table using the remaining bits of the VPN:



This **page-table index** (`PTIndex` for short) can then be used to index into the page table itself, giving us the address of our PTE:

```
PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
```

Note that the page-frame number (PFN) obtained from the page-directory entry must be left-shifted into place before combining it with the page-table index to form the address of the PTE.

To see if this all makes sense, we'll now fill in a multi-level page table with some actual values, and translate a single virtual address. Let's begin with the **page directory** for this example (left side of Figure 20.5).

In the figure, you can see that each page directory entry (PDE) describes something about a page of the page table for the address space. In this example, we have two valid regions in the address space (at the beginning and end), and a number of invalid mappings in-between.

In physical page 100 (the physical frame number of the 0th page of the page table), we have the first page of 16 page table entries for the first 16 VPNs in the address space. See Figure 20.5 (middle part) for the contents of this portion of the page table.

Page Directory		**Page of PT (@PFN:100)**			**Page of PT (@PFN:101)**		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	55	1	rw-
101	1	—	0	—	45	1	rw-

Figure 20.5: **A Page Directory, And Pieces Of Page Table**

This page of the page table contains the mappings for the first 16 VPNs; in our example, VPNs 0 and 1 are valid (the code segment), as are 4 and 5 (the heap). Thus, the table has mapping information for each of those pages. The rest of the entries are marked invalid.

The other valid page of the page table is found inside PFN 101. This page contains mappings for the last 16 VPNs of the address space; see Figure 20.5 (right) for details.

In the example, VPNs 254 and 255 (the stack) have valid mappings. Hopefully, what we can see from this example is how much space savings are possible with a multi-level indexed structure. In this example, instead of allocating the full *sixteen* pages for a linear page table, we allocate only *three*: one for the page directory, and two for the chunks of the page table that have valid mappings. The savings for large (32-bit or 64-bit) address spaces could obviously be much greater.

Finally, let's use this information in order to perform a translation. Here is an address that refers to the 0th byte of VPN 254: `0x3F80`, or `11 1111 1000 0000` in binary.

Recall that we will use the top 4 bits of the VPN to index into the page directory. Thus, `1111` will choose the last (15th, if you start at the 0th) entry of the page directory above. This points us to a valid page of the page table located at address `101`. We then use the next 4 bits of the VPN (`1110`) to index into that page of the page table and find the desired PTE. 1110 is the next-to-last (14th) entry on the page, and tells us that page 254 of our virtual address space is mapped at physical page 55. By concatenating PFN=55 (or hex `0x37`) with offset=000000, we can thus form our desired physical address and issue the request to the memory system: `PhysAddr = (PTE.PFN << SHIFT) + offset = 00 1101 1100 0000 = 0x0DC0`.

You should now have some idea of how to construct a two-level page table, using a page directory which points to pages of the page table. Unfortunately, however, our work is not done. As we'll now discuss, sometimes two levels of page table is not enough!

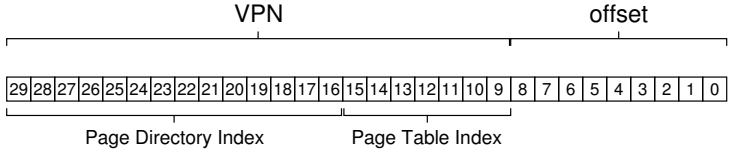
More Than Two Levels

In our example thus far, we've assumed that multi-level page tables only have two levels: a page directory and then pieces of the page table. In some cases, a deeper tree is possible (and indeed, needed).

Let's take a simple example and use it to show why a deeper multi-level table can be useful. In this example, assume we have a 30-bit virtual address space, and a small (512 byte) page. Thus our virtual address has a 21-bit virtual page number component and a 9-bit offset.

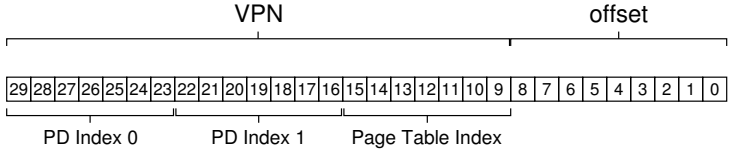
Remember our goal in constructing a multi-level page table: to make each piece of the page table fit within a single page. Thus far, we've only considered the page table itself; however, what if the page directory gets too big?

To determine how many levels are needed in a multi-level table to make all pieces of the page table fit within a page, we start by determining how many page-table entries fit within a page. Given our page size of 512 bytes, and assuming a PTE size of 4 bytes, you should see that you can fit 128 PTEs on a single page. When we index into a page of the page table, we can thus conclude we'll need the least significant 7 bits ($log_2 128$) of the VPN as an index:



What you also might notice from the diagram above is how many bits are left into the (large) page directory: 14. If our page directory has 2^{14} entries, it spans not one page but 128, and thus our goal of making every piece of the multi-level page table fit into a page vanishes.

To remedy this problem, we build a further level of the tree, by splitting the page directory itself into multiple pages, and then adding another page directory on top of that, to point to the pages of the page directory. We can thus split up our virtual address as follows:



Now, when indexing the upper-level page directory, we use the very top bits of the virtual address (`PD Index 0` in the diagram); this index can be used to fetch the page-directory entry from the top-level page directory. If valid, the second level of the page directory is consulted by combining the physical frame number from the top-level PDE and the next part of the VPN (`PD Index 1`). Finally, if valid, the PTE address can be formed by using the page-table index combined with the address from the second-level PDE. Whew! That's a lot of work. And all just to look something up in a multi-level table.

The Translation Process: Remember the TLB

To summarize the entire process of address translation using a two-level page table, we once again present the control flow in algorithmic form (Figure 20.6). The figure shows what happens in hardware (assuming a hardware-managed TLB) upon *every* memory reference.

As you can see from the figure, before any of the complicated multi-level page table access occurs, the hardware first checks the TLB; upon

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)   // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset   = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10 else                  // TLB Miss
11     // first, get page directory entry
12     PDIndex = (VPN & PD_MASK) >> PD_SHIFT
13     PDEAddr = PDBR + (PDIndex * sizeof(PDE))
14     PDE     = AccessMemory(PDEAddr)
15     if (PDE.Valid == False)
16         RaiseException(SEGMENTATION_FAULT)
17     else
18         // PDE is valid: now fetch PTE from page table
19         PTIndex = (VPN & PT_MASK) >> PT_SHIFT
20         PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
21         PTE     = AccessMemory(PTEAddr)
22         if (PTE.Valid == False)
23             RaiseException(SEGMENTATION_FAULT)
24         else if (CanAccess(PTE.ProtectBits) == False)
25             RaiseException(PROTECTION_FAULT)
26         else
27             TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
28             RetryInstruction()
```

Figure 20.6: **Multi-level Page Table Control Flow**

a hit, the physical address is formed directly *without* accessing the page table at all, as before. Only upon a TLB miss does the hardware need to perform the full multi-level lookup. On this path, you can see the cost of our traditional two-level page table: two additional memory accesses to look up a valid translation.

20.4 Inverted Page Tables

An even more extreme space savings in the world of page tables is found with **inverted page tables**. Here, instead of having many page tables (one per process of the system), we keep a single page table that has an entry for each *physical page* of the system. The entry tells us which process is using this page, and which virtual page of that process maps to this physical page.

Finding the correct entry is now a matter of searching through this data structure. A linear scan would be expensive, and thus a hash table is often built over the base structure to speed lookups. The PowerPC is one example of such an architecture [JM98].

More generally, inverted page tables illustrate what we've said from the beginning: page tables are just data structures. You can do lots of crazy things with data structures, making them smaller or bigger, making them slower or faster. Multi-level and inverted page tables are just two examples of the many things one could do.

20.5 Swapping the Page Tables to Disk

Finally, we discuss the relaxation of one final assumption. Thus far, we have assumed that page tables reside in kernel-owned physical memory. Even with our many tricks to reduce the size of page tables, it is still possible, however, that they may be too big to fit into memory all at once. Thus, some systems place such page tables in **kernel virtual memory**, thereby allowing the system to **swap** some of these page tables to disk when memory pressure gets a little tight. We'll talk more about this in a future chapter (namely, the case study on VAX/VMS), once we understand how to move pages in and out of memory in more detail.

20.6 Summary

We have now seen how real page tables are built; not necessarily just as linear arrays but as more complex data structures. The trade-offs such tables present are in time and space — the bigger the table, the faster a TLB miss can be serviced, as well as the converse — and thus the right choice of structure depends strongly on the constraints of the given environment.

In a memory-constrained system (like many older systems), small structures make sense; in a system with a reasonable amount of memory and with workloads that actively use a large number of pages, a bigger table that speeds up TLB misses might be the right choice. With software-managed TLBs, the entire space of data structures opens up to the delight of the operating system innovator (hint: that's you). What new structures can you come up with? What problems do they solve? Think of these questions as you fall asleep, and dream the big dreams that only operating-system developers can dream.

References

[BOH10] "Computer Systems: A Programmer's Perspective"
Randal E. Bryant and David R. O'Hallaron
Addison-Wesley, 2010
We have yet to find a good first reference to the multi-level page table. However, this great textbook by Bryant and O'Hallaron dives into the details of x86, which at least is an early system that used such structures. It's also just a great book to have.

[JM98] "Virtual Memory: Issues of Implementation"
Bruce Jacob and Trevor Mudge
IEEE Computer, June 1998
An excellent survey of a number of different systems and their approach to virtualizing memory. Plenty of details on x86, PowerPC, MIPS, and other architectures.

[LL82] "Virtual Memory Management in the VAX/VMS Operating System"
Hank Levy and P. Lipman
IEEE Computer, Vol. 15, No. 3, March 1982
A terrific paper about a real virtual memory manager in a classic operating system, VMS. So terrific, in fact, that we'll use it to review everything we've learned about virtual memory thus far a few chapters from now.

[M28] "Reese's Peanut Butter Cups"
Mars Candy Corporation.
Apparently these fine confections were invented in 1928 by Harry Burnett Reese, a former dairy farmer and shipping foreman for one Milton S. Hershey. At least, that is what it says on Wikipedia. If true, Hershey and Reese probably hated each other's guts, as any two chocolate barons should.

[N+02] "Practical, Transparent Operating System Support for Superpages"
Juan Navarro, Sitaram Iyer, Peter Druschel, Alan Cox
OSDI '02, Boston, Massachusetts, October 2002
*A nice paper showing all the details you have to get right to incorporate large pages, or **superpages**, into a modern OS. Not as easy as you might think, alas.*

[M07] "Multics: History"
Available: `http://www.multicians.org/history.html`
This amazing web site provides a huge amount of history on the Multics system, certainly one of the most influential systems in OS history. The quote from therein: "Jack Dennis of MIT contributed influential architectural ideas to the beginning of Multics, especially the idea of combining paging and segmentation." (from Section 1.2.1)

Homework

This fun little homework tests if you understand how a multi-level page table works. And yes, there is some debate over the use of the term "fun" in the previous sentence. The program is called, perhaps unsurprisingly: `paging-multilevel-translate.py`; see the README for details.

Questions

1. With a linear page table, you need a single register to locate the page table, assuming that hardware does the lookup upon a TLB miss. How many registers do you need to locate a two-level page table? A three-level table?
2. Use the simulator to perform translations given random seeds 0, 1, and 2, and check your answers using the `-c` flag. How many memory references are needed to perform each lookup?
3. Given your understanding of how cache memory works, how do you think memory references to the page table will behave in the cache? Will they lead to lots of cache hits (and thus fast accesses?) Or lots of misses (and thus slow accesses)?

21

Beyond Physical Memory: Mechanisms

Thus far, we've assumed that an address space is unrealistically small and fits into physical memory. In fact, we've been assuming that *every* address space of every running process fits into memory. We will now relax these big assumptions, and assume that we wish to support many concurrently-running large address spaces.

To do so, we require an additional level in the **memory hierarchy**. Thus far, we have assumed that all pages reside in physical memory. However, to support large address spaces, the OS will need a place to stash away portions of address spaces that currently aren't in great demand. In general, the characteristics of such a location are that it should have more capacity than memory; as a result, it is generally slower (if it were faster, we would just use it as memory, no?). In modern systems, this role is usually served by a **hard disk drive**. Thus, in our memory hierarchy, big and slow hard drives sit at the bottom, with memory just above. And thus we arrive at the crux of the problem:

THE CRUX: HOW TO GO BEYOND PHYSICAL MEMORY
How can the OS make use of a larger, slower device to transparently provide the illusion of a large virtual address space?

One question you might have: why do we want to support a single large address space for a process? Once again, the answer is convenience and ease of use. With a large address space, you don't have to worry about if there is room enough in memory for your program's data structures; rather, you just write the program naturally, allocating memory as needed. It is a powerful illusion that the OS provides, and makes your life vastly simpler. You're welcome! A contrast is found in older systems that used **memory overlays**, which required programmers to manually move pieces of code or data in and out of memory as they were needed [D97]. Try imagining what this would be like: before calling a function or accessing some data, you need to first arrange for the code or data to be in memory; yuck!

ASIDE: **STORAGE TECHNOLOGIES**
We'll delve much more deeply into how I/O devices actually work later (see the chapter on I/O devices). So be patient! And of course the slower device need not be a hard disk, but could be something more modern such as a Flash-based SSD. We'll talk about those things too. For now, just assume we have a big and relatively-slow device which we can use to help us build the illusion of a very large virtual memory, even bigger than physical memory itself.

Beyond just a single process, the addition of swap space allows the OS to support the illusion of a large virtual memory for multiple concurrently-running processes. The invention of multiprogramming (running multiple programs "at once", to better utilize the machine) almost demanded the ability to swap out some pages, as early machines clearly could not hold all the pages needed by all processes at once. Thus, the combination of multiprogramming and ease-of-use leads us to want to support using more memory than is physically available. It is something that all modern VM systems do; it is now something we will learn more about.

21.1 Swap Space

The first thing we will need to do is to reserve some space on the disk for moving pages back and forth. In operating systems, we generally refer to such space as **swap space**, because we *swap* pages out of memory to it and *swap* pages into memory from it. Thus, we will simply assume that the OS can read from and write to the swap space, in page-sized units. To do so, the OS will need to remember the **disk address** of a given page.

The size of the swap space is important, as ultimately it determines the maximum number of memory pages that can be in use by a system at a given time. Let us assume for simplicity that it is *very* large for now.

In the tiny example (Figure 21.1), you can see a little example of a 4-page physical memory and an 8-page swap space. In the example, three processes (Proc 0, Proc 1, and Proc 2) are actively sharing physical memory; each of the three, however, only have some of their valid pages in memory, with the rest located in swap space on disk. A fourth process (Proc 3) has all of its pages swapped out to disk, and thus clearly isn't currently running. One block of swap remains free. Even from this tiny example, hopefully you can see how using swap space allows the system to pretend that memory is larger than it actually is.

We should note that swap space is not the only on-disk location for swapping traffic. For example, assume you are running a program binary (e.g., `ls`, or your own compiled `main` program). The code pages from this binary are initially found on disk, and when the program runs, they are loaded into memory (either all at once when the program starts execution,

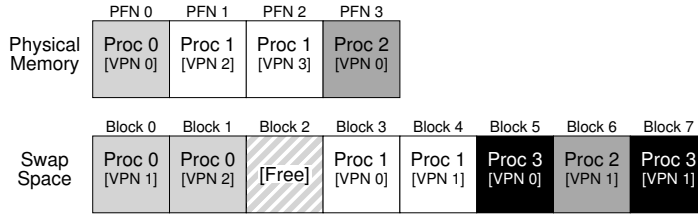


Figure 21.1: **Physical Memory and Swap Space**

or, as in modern systems, one page at a time when needed). However, if the system needs to make room in physical memory for other needs, it can safely re-use the memory space for these code pages, knowing that it can later swap them in again from the on-disk binary in the file system.

21.2 The Present Bit

Now that we have some space on the disk, we need to add some machinery higher up in the system in order to support swapping pages to and from the disk. Let us assume, for simplicity, that we have a system with a hardware-managed TLB.

Recall first what happens on a memory reference. The running process generates virtual memory references (for instruction fetches, or data accesses), and, in this case, the hardware translates them into physical addresses before fetching the desired data from memory.

Remember that the hardware first extracts the VPN from the virtual address, checks the TLB for a match (a **TLB hit**), and if a hit, produces the resulting physical address and fetches it from memory. This is hopefully the common case, as it is fast (requiring no additional memory accesses).

If the VPN is not found in the TLB (i.e., a **TLB miss**), the hardware locates the page table in memory (using the **page table base register**) and looks up the **page table entry (PTE)** for this page using the VPN as an index. If the page is valid and present in physical memory, the hardware extracts the PFN from the PTE, installs it in the TLB, and retries the instruction, this time generating a TLB hit; so far, so good.

If we wish to allow pages to be swapped to disk, however, we must add even more machinery. Specifically, when the hardware looks in the PTE, it may find that the page is *not present* in physical memory. The way the hardware (or the OS, in a software-managed TLB approach) determines this is through a new piece of information in each page-table entry, known as the **present bit**. If the present bit is set to one, it means the page is present in physical memory and everything proceeds as above; if it is set to zero, the page is *not* in memory but rather on disk somewhere. The act of accessing a page that is not in physical memory is commonly referred to as a **page fault**.

ASIDE: **SWAPPING TERMINOLOGY AND OTHER THINGS**
Terminology in virtual memory systems can be a little confusing and variable across machines and operating systems. For example, a **page fault** more generally could refer to any reference to a page table that generates a fault of some kind: this could include the type of fault we are discussing here, i.e., a page-not-present fault, but sometimes can refer to illegal memory accesses. Indeed, it is odd that we call what is definitely a legal access (to a page mapped into the virtual address space of a process, but simply not in physical memory at the time) a "fault" at all; really, it should be called a **page miss**. But often, when people say a program is "page faulting", they mean that it is accessing parts of its virtual address space that the OS has swapped out to disk.

We suspect the reason that this behavior became known as a "fault" relates to the machinery in the operating system to handle it. When something unusual happens, i.e., when something the hardware doesn't know how to handle occurs, the hardware simply transfers control to the OS, hoping it can make things better. In this case, a page that a process wants to access is missing from memory; the hardware does the only thing it can, which is raise an exception, and the OS takes over from there. As this is identical to what happens when a process does something illegal, it is perhaps not surprising that we term the activity a "fault."

Upon a page fault, the OS is invoked to service the page fault. A particular piece of code, known as a **page-fault handler**, runs, and must service the page fault, as we now describe.

21.3 The Page Fault

Recall that with TLB misses, we have two types of systems: hardware-managed TLBs (where the hardware looks in the page table to find the desired translation) and software-managed TLBs (where the OS does). In either type of system, if a page is not present, the OS is put in charge to handle the page fault. The appropriately-named OS **page-fault handler** runs to determine what to do. Virtually all systems handle page faults in software; even with a hardware-managed TLB, the hardware trusts the OS to manage this important duty.

If a page is not present and has been swapped to disk, the OS will need to swap the page into memory in order to service the page fault. Thus, a question arises: how will the OS know where to find the desired page? In many systems, the page table is a natural place to store such information. Thus, the OS could use the bits in the PTE normally used for data such as the PFN of the page for a disk address. When the OS receives a page fault for a page, it looks in the PTE to find the address, and issues the request to disk to fetch the page into memory.

ASIDE: **WHY HARDWARE DOESN'T HANDLE PAGE FAULTS**
We know from our experience with the TLB that hardware designers are loathe to trust the OS to do much of anything. So why do they trust the OS to handle a page fault? There are a few main reasons. First, page faults to disk are *slow*; even if the OS takes a long time to handle a fault, executing tons of instructions, the disk operation itself is traditionally so slow that the extra overheads of running software are minimal. Second, to be able to handle a page fault, the hardware would have to understand swap space, how to issue I/Os to the disk, and a lot of other details which it currently doesn't know much about. Thus, for both reasons of performance and simplicity, the OS handles page faults, and even hardware types can be happy.

When the disk I/O completes, the OS will then update the page table to mark the page as present, update the PFN field of the page-table entry (PTE) to record the in-memory location of the newly-fetched page, and retry the instruction. This next attempt may generate a TLB miss, which would then be serviced and update the TLB with the translation (one could alternately update the TLB when servicing the page fault to avoid this step). Finally, a last restart would find the translation in the TLB and thus proceed to fetch the desired data or instruction from memory at the translated physical address.

Note that while the I/O is in flight, the process will be in the **blocked** state. Thus, the OS will be free to run other ready processes while the page fault is being serviced. Because I/O is expensive, this **overlap** of the I/O (page fault) of one process and the execution of another is yet another way a multiprogrammed system can make the most effective use of its hardware.

21.4 What If Memory Is Full?

In the process described above, you may notice that we assumed there is plenty of free memory in which to **page in** a page from swap space. Of course, this may not be the case; memory may be full (or close to it). Thus, the OS might like to first **page out** one or more pages to make room for the new page(s) the OS is about to bring in. The process of picking a page to kick out, or **replace** is known as the **page-replacement policy**.

As it turns out, a lot of thought has been put into creating a good page-replacement policy, as kicking out the wrong page can exact a great cost on program performance. Making the wrong decision can cause a program to run at disk-like speeds instead of memory-like speeds; in current technology that means a program could run 10,000 or 100,000 times slower. Thus, such a policy is something we should study in some detail; indeed, that is exactly what we will do in the next chapter. For now, it is good enough to understand that such a policy exists, built on top of the mechanisms described here.

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)   // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset   = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10 else                  // TLB Miss
11     PTEAddr = PTBR + (VPN * sizeof(PTE))
12     PTE = AccessMemory(PTEAddr)
13     if (PTE.Valid == False)
14         RaiseException(SEGMENTATION_FAULT)
15     else
16         if (CanAccess(PTE.ProtectBits) == False)
17             RaiseException(PROTECTION_FAULT)
18         else if (PTE.Present == True)
19             // assuming hardware-managed TLB
20             TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
21             RetryInstruction()
22         else if (PTE.Present == False)
23             RaiseException(PAGE_FAULT)
```

Figure 21.2: **Page-Fault Control Flow Algorithm (Hardware)**

21.5 Page Fault Control Flow

With all of this knowledge in place, we can now roughly sketch the complete control flow of memory access. In other words, when somebody asks you "what happens when a program fetches some data from memory?", you should have a pretty good idea of all the different possibilities. See the control flow in Figures 21.2 and 21.3 for more details; the first figure shows what the hardware does during translation, and the second what the OS does upon a page fault.

From the hardware control flow diagram in Figure 21.2, notice that there are now three important cases to understand when a TLB miss occurs. First, that the page was both **present** and **valid** (Lines 18–21); in this case, the TLB miss handler can simply grab the PFN from the PTE, retry the instruction (this time resulting in a TLB hit), and thus continue as described (many times) before. In the second case (Lines 22–23), the page fault handler must be run; although this was a legitimate page for the process to access (it is valid, after all), it is not present in physical memory. Third (and finally), the access could be to an invalid page, due for example to a bug in the program (Lines 13–14). In this case, no other bits in the PTE really matter; the hardware traps this invalid access, and the OS trap handler runs, likely terminating the offending process.

From the software control flow in Figure 21.3, we can see what the OS roughly must do in order to service the page fault. First, the OS must find a physical frame for the soon-to-be-faulted-in page to reside within; if there is no such page, we'll have to wait for the replacement algorithm to run and kick some pages out of memory, thus freeing them for use here.

```
1  PFN = FindFreePhysicalPage()
2  if (PFN == -1)              // no free page found
3      PFN = EvictPage()       // run replacement algorithm
4  DiskRead(PTE.DiskAddr, pfn) // sleep (waiting for I/O)
5  PTE.present = True          // update page table with present
6  PTE.PFN     = PFN           // bit and translation (PFN)
7  RetryInstruction()          // retry instruction
```

Figure 21.3: **Page-Fault Control Flow Algorithm (Software)**

With a physical frame in hand, the handler then issues the I/O request to read in the page from swap space. Finally, when that slow operation completes, the OS updates the page table and retries the instruction. The retry will result in a TLB miss, and then, upon another retry, a TLB hit, at which point the hardware will be able to access the desired item.

21.6 When Replacements Really Occur

Thus far, the way we've described how replacements occur assumes that the OS waits until memory is entirely full, and only then replaces (evicts) a page to make room for some other page. As you can imagine, this is a little bit unrealistic, and there are many reasons for the OS to keep a small portion of memory free more proactively.

To keep a small amount of memory free, most operating systems thus have some kind of **high watermark** (HW) and **low watermark** (LW) to help decide when to start evicting pages from memory. How this works is as follows: when the OS notices that there are fewer than LW pages available, a background thread that is responsible for freeing memory runs. The thread evicts pages until there are HW pages available. The background thread, sometimes called the **swap daemon** or **page daemon**[1], then goes to sleep, happy that it has freed some memory for running processes and the OS to use.

By performing a number of replacements at once, new performance optimizations become possible. For example, many systems will **cluster** or **group** a number of pages and write them out at once to the swap partition, thus increasing the efficiency of the disk [LL82]; as we will see later when we discuss disks in more detail, such clustering reduces seek and rotational overheads of a disk and thus increases performance noticeably.

To work with the background paging thread, the control flow in Figure 21.3 should be modified slightly; instead of performing a replacement directly, the algorithm would instead simply check if there are any free pages available. If not, it would inform the background paging thread that free pages are needed; when the thread frees up some pages, it would re-awaken the original thread, which could then page in the desired page and go about its work.

[1] The word "daemon", usually pronounced "demon", is an old term for a background thread or process that does something useful. Turns out (once again!) that the source of the term is Multics [CS94].

TIP: DO WORK IN THE BACKGROUND

When you have some work to do, it is often a good idea to do it in the **background** to increase efficiency and to allow for grouping of operations. Operating systems often do work in the background; for example, many systems buffer file writes in memory before actually writing the data to disk. Doing so has many possible benefits: increased disk efficiency, as the disk may now receive many writes at once and thus better be able to schedule them; improved latency of writes, as the application thinks the writes completed quite quickly; the possibility of work reduction, as the writes may need never to go to disk (i.e., if the file is deleted); and better use of **idle time**, as the background work may possibly be done when the system is otherwise idle, thus better utilizing the hardware [G+95].

21.7 Summary

In this brief chapter, we have introduced the notion of accessing more memory than is physically present within a system. To do so requires more complexity in page-table structures, as a **present bit** (of some kind) must be included to tell us whether the page is present in memory or not. When not, the operating system **page-fault handler** runs to service the **page fault**, and thus arranges for the transfer of the desired page from disk to memory, perhaps first replacing some pages in memory to make room for those soon to be swapped in.

Recall, importantly (and amazingly!), that these actions all take place **transparently** to the process. As far as the process is concerned, it is just accessing its own private, contiguous virtual memory. Behind the scenes, pages are placed in arbitrary (non-contiguous) locations in physical memory, and sometimes they are not even present in memory, requiring a fetch from disk. While we hope that in the common case a memory access is fast, in some cases it will take multiple disk operations to service it; something as simple as performing a single instruction can, in the worst case, take many milliseconds to complete.

References

[CS94] "Take Our Word For It"
F. Corbato and R. Steinberg
Available: `http://www.takeourword.com/TOW146/page4.html`
Richard Steinberg writes: "Someone has asked me the origin of the word daemon as it applies to computing. Best I can tell based on my research, the word was first used by people on your team at Project MAC using the IBM 7094 in 1963." Professor Corbato replies: "Our use of the word daemon was inspired by the Maxwell's daemon of physics and thermodynamics (my background is in physics). Maxwell's daemon was an imaginary agent which helped sort molecules of different speeds and worked tirelessly in the background. We fancifully began to use the word daemon to describe background processes which worked tirelessly to perform system chores."

[D97] "Before Memory Was Virtual"
Peter Denning
From *In the Beginning: Recollections of Software Pioneers*, Wiley, November 1997
An excellent historical piece by one of the pioneers of virtual memory and working sets.

[G+95] "Idleness is not sloth"
Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, John Wilkes
USENIX ATC '95, New Orleans, Louisiana
A fun and easy-to-read discussion of how idle time can be better used in systems, with lots of good examples.

[LL82] "Virtual Memory Management in the VAX/VMS Operating System"
Hank Levy and P. Lipman
IEEE Computer, Vol. 15, No. 3, March 1982
Not the first place where such clustering was used, but a clear and simple explanation of how such a mechanism works.

Homework (Measurement)

This homework introduces you to a new tool, **vmstat**, and how it can be used to get a sense of what your computer is doing with regards to memory, CPU, and I/O usage (with a focus on memory and swapping).

Read the associated README and examine the code in `mem.c` before proceeding to the exercises and questions below.

Questions

1. First, open two separate terminal connections to the *same* machine, so that you can easily run something in one window and the other.

   Now, in one window, run `vmstat 1`, which shows statistics about machine usage every second. Read the man page, the associated README, and any other information you need so that you can understand its output. Leave this window running `vmstat` for the rest of the exercises below.

   Now, we will run the program `mem.c` but with very little memory usage. This can be accomplished by typing `./mem 1` (which uses only 1 MB of memory). How do the CPU usage statistics change when running `mem`? Do the numbers in the `user time` column make sense? How does this change when running more than one instance of `mem` at once?

2. Let's now start looking at some of the memory statistics while running `mem`. We'll focus on two columns: `swpd` (the amount of virtual memory used) and `free` (the amount of idle memory). Run `./mem 1024` (which allocates 1024 MB) and watch how these values change. Then kill the running program (by typing control-c) and watch again how the values change. What do you notice about the values? In particular, how does the `free` column change when the program exits? Does the amount of free memory increase by the expected amount when `mem` exits?

3. We'll next look at the `swap` columns (`si` and `so`), which indicate how much swapping is taking place to and from the disk. Of course, to activate these, you'll need to run `mem` with large amounts of memory. First, examine how much free memory is on your Linux system (for example, by typing `cat /proc/meminfo`; type `man proc` for details on the `/proc` file system and the types of information you can find there). One of the first entries in `/proc/meminfo` is the total amount of memory in your system. Let's assume it's something like 8 GB of memory; if so, start by running `mem 4000` (about 4 GB) and watching the swap in/out columns. Do they ever give non-zero values? Then, try with `5000`, `6000`, etc. What happens to these values as the program enters the second loop (and beyond), as compared to the first loop? How much data (total)

are swapped in and out during the second, third, and subsequent loops? (do the numbers make sense?)

4. Do the same experiments as above, but now watch the other statistics (such as CPU utilization, and block I/O statistics). How do they change when `mem` is running?

5. Now let's examine performance. Pick an input for `mem` that comfortably fits in memory (say `4000` if the amount of memory on the system is 8 GB). How long does loop 0 take (and subsequent loops 1, 2, etc.)? Now pick a size comfortably beyond the size of memory (say `12000` again assuming 8 GB of memory). How long do the loops take here? How do the bandwidth numbers compare? How different is performance when constantly swapping versus fitting everything comfortably in memory? Can you make a graph, with the size of memory used by `mem` on the x-axis, and the bandwidth of accessing said memory on the y-axis? Finally, how does the performance of the first loop compare to that of subsequent loops, for both the case where everything fits in memory and where it doesn't?

6. Swap space isn't infinite. You can use the tool `swapon` with the `-s` flag to see how much swap space is available. What happens if you try to run `mem` with increasingly large values, beyond what seems to be available in swap? At what point does the memory allocation fail?

7. Finally, if you're advanced, you can configure your system to use different swap devices using `swapon` and `swapoff`. Read the man pages for details. If you have access to different hardware, see how the performance of swapping changes when swapping to a classic hard drive, a flash-based SSD, and even a RAID array. How much can swapping performance be improved via newer devices? How close can you get to in-memory performance?

22

Beyond Physical Memory: Policies

In a virtual memory manager, life is easy when you have a lot of free memory. A page fault occurs, you find a free page on the free-page list, and assign it to the faulting page. Hey, Operating System, congratulations! You did it again.

Unfortunately, things get a little more interesting when little memory is free. In such a case, this **memory pressure** forces the OS to start **paging out** pages to make room for actively-used pages. Deciding which page (or pages) to **evict** is encapsulated within the **replacement policy** of the OS; historically, it was one of the most important decisions the early virtual memory systems made, as older systems had little physical memory. Minimally, it is an interesting set of policies worth knowing a little more about. And thus our problem:

THE CRUX: HOW TO DECIDE WHICH PAGE TO EVICT
How can the OS decide which page (or pages) to evict from memory? This decision is made by the replacement policy of the system, which usually follows some general principles (discussed below) but also includes certain tweaks to avoid corner-case behaviors.

22.1 Cache Management

Before diving into policies, we first describe the problem we are trying to solve in more detail. Given that main memory holds some subset of all the pages in the system, it can rightly be viewed as a **cache** for virtual memory pages in the system. Thus, our goal in picking a replacement policy for this cache is to minimize the number of **cache misses**, i.e., to minimize the number of times that we have to fetch a page from disk. Alternately, one can view our goal as maximizing the number of **cache hits**, i.e., the number of times a page that is accessed is found in memory.

Knowing the number of cache hits and misses let us calculate the **average memory access time** (**AMAT**) for a program (a metric computer

architects compute for hardware caches [HP06]). Specifically, given these values, we can compute the AMAT of a program as follows:

$$AMAT = (P_{Hit} \cdot T_M) + (P_{Miss} \cdot T_D) \quad (22.1)$$

where T_M represents the cost of accessing memory, T_D the cost of accessing disk, P_{Hit} the probability of finding the data item in the cache (a hit), and P_{Miss} the probability of not finding the data in the cache (a miss). P_{Hit} and P_{Miss} each vary from 0.0 to 1.0, and $P_{Miss} + P_{Hit} = 1.0$.

For example, let us imagine a machine with a (tiny) address space: 4KB, with 256-byte pages. Thus, a virtual address has two components: a 4-bit VPN (the most-significant bits) and an 8-bit offset (the least-significant bits). Thus, a process in this example can access 2^4 or 16 total virtual pages. In this example, the process generates the following memory references (i.e., virtual addresses): 0x000, 0x100, 0x200, 0x300, 0x400, 0x500, 0x600, 0x700, 0x800, 0x900. These virtual addresses refer to the first byte of each of the first ten pages of the address space (the page number being the first hex digit of each virtual address).

Let us further assume that every page except virtual page 3 is already in memory. Thus, our sequence of memory references will encounter the following behavior: hit, hit, hit, miss, hit, hit, hit, hit, hit, hit. We can compute the **hit rate** (the percent of references found in memory): 90% ($P_{Hit} = 0.9$), as 9 out of 10 references are in memory. The **miss rate** is obviously 10% ($P_{Miss} = 0.1$).

To calculate AMAT, we simply need to know the cost of accessing memory and the cost of accessing disk. Assuming the cost of accessing memory (T_M) is around 100 nanoseconds, and the cost of accessing disk (T_D) is about 10 milliseconds, we have the following AMAT: $0.9 \cdot 100ns + 0.1 \cdot 10ms$, which is $90ns + 1ms$, or 1.00009 ms, or about 1 millisecond. If our hit rate had instead been 99.9%, the result is quite different: AMAT is 10.1 microseconds, or roughly 100 times faster. As the hit rate approaches 100%, AMAT approaches 100 nanoseconds.

Unfortunately, as you can see in this example, the cost of disk access is so high in modern systems that even a tiny miss rate will quickly dominate the overall AMAT of running programs. Clearly, we need to avoid as many misses as possible or run slowly, at the rate of the disk. One way to help with this is to carefully develop a smart policy, as we now do.

22.2 The Optimal Replacement Policy

To better understand how a particular replacement policy works, it would be nice to compare it to the best possible replacement policy. As it turns out, such an **optimal** policy was developed by Belady many years ago [B66] (he originally called it MIN). The optimal replacement policy leads to the fewest number of misses overall. Belady showed that a simple (but, unfortunately, difficult to implement!) approach that replaces the page that will be accessed *furthest in the future* is the optimal policy, resulting in the fewest-possible cache misses.

TIP: COMPARING AGAINST OPTIMAL IS USEFUL
Although optimal is not very practical as a real policy, it is incredibly useful as a comparison point in simulation or other studies. Saying that your fancy new algorithm has a 80% hit rate isn't meaningful in isolation; saying that optimal achieves an 82% hit rate (and thus your new approach is quite close to optimal) makes the result more meaningful and gives it context. Thus, in any study you perform, knowing what the optimal is lets you perform a better comparison, showing how much improvement is still possible, and also when you can *stop* making your policy better, because it is close enough to the ideal [AD03].

Hopefully, the intuition behind the optimal policy makes sense. Think about it like this: if you have to throw out some page, why not throw out the one that is needed the furthest from now? By doing so, you are essentially saying that all the other pages in the cache are more important than the one furthest out. The reason this is true is simple: you will refer to the other pages before you refer to the one furthest out.

Let's trace through a simple example to understand the decisions the optimal policy makes. Assume a program accesses the following stream of virtual pages: 0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1. Figure 22.1 shows the behavior of optimal, assuming a cache that fits three pages.

In the figure, you can see the following actions. Not surprisingly, the first three accesses are misses, as the cache begins in an empty state; such a miss is sometimes referred to as a **cold-start miss** (or **compulsory miss**). Then we refer again to pages 0 and 1, which both hit in the cache. Finally, we reach another miss (to page 3), but this time the cache is full; a replacement must take place! Which begs the question: which page should we replace? With the optimal policy, we examine the future for each page currently in the cache (0, 1, and 2), and see that 0 is accessed almost immediately, 1 is accessed a little later, and 2 is accessed furthest in the future. Thus the optimal policy has an easy choice: evict page 2, resulting in pages 0, 1, and 3 in the cache. The next three references are hits, but then

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

Figure 22.1: **Tracing The Optimal Policy**

ASIDE: **TYPES OF CACHE MISSES**
In the computer architecture world, architects sometimes find it useful to characterize misses by type, into one of three categories: compulsory, capacity, and conflict misses, sometimes called the **Three C's** [H87]. A **compulsory miss** (or **cold-start miss** [EF78]) occurs because the cache is empty to begin with and this is the first reference to the item; in contrast, a **capacity miss** occurs because the cache ran out of space and had to evict an item to bring a new item into the cache. The third type of miss (a **conflict miss**) arises in hardware because of limits on where an item can be placed in a hardware cache, due to something known as **set-associativity**; it does not arise in the OS page cache because such caches are always **fully-associative**, i.e., there are no restrictions on where in memory a page can be placed. See H&P for details [HP06].

we get to page 2, which we evicted long ago, and suffer another miss. Here the optimal policy again examines the future for each page in the cache (0, 1, and 3), and sees that as long as it doesn't evict page 1 (which is about to be accessed), we'll be OK. The example shows page 3 getting evicted, although 0 would have been a fine choice too. Finally, we hit on page 1 and the trace completes.

We can also calculate the hit rate for the cache: with 6 hits and 5 misses, the hit rate is $\frac{Hits}{Hits+Misses}$ which is $\frac{6}{6+5}$ or 54.5%. You can also compute the hit rate *modulo* compulsory misses (i.e., ignore the *first* miss to a given page), resulting in a 85.7% hit rate.

Unfortunately, as we saw before in the development of scheduling policies, the future is not generally known; you can't build the optimal policy for a general-purpose operating system[1]. Thus, in developing a real, deployable policy, we will focus on approaches that find some other way to decide which page to evict. The optimal policy will thus serve only as a comparison point, to know how close we are to "perfect".

22.3 A Simple Policy: FIFO

Many early systems avoided the complexity of trying to approach optimal and employed very simple replacement policies. For example, some systems used **FIFO** (first-in, first-out) replacement, where pages were simply placed in a queue when they enter the system; when a replacement occurs, the page on the tail of the queue (the "first-in" page) is evicted. FIFO has one great strength: it is quite simple to implement.

Let's examine how FIFO does on our example reference stream (Figure 22.2, page 5). We again begin our trace with three compulsory misses to pages 0, 1, and 2, and then hit on both 0 and 1. Next, page 3 is referenced, causing a miss; the replacement decision is easy with FIFO: pick the page

[1] If you can, let us know! We can become rich together. Or, like the scientists who "discovered" cold fusion, widely scorned and mocked [FP89].

Access	Hit/Miss?	Evict	Resulting Cache State	
0	Miss		First-in→	0
1	Miss		First-in→	0, 1
2	Miss		First-in→	0, 1, 2
0	Hit		First-in→	0, 1, 2
1	Hit		First-in→	0, 1, 2
3	Miss	0	First-in→	1, 2, 3
0	Miss	1	First-in→	2, 3, 0
3	Hit		First-in→	2, 3, 0
1	Miss	2	First-in→	3, 0, 1
2	Miss	3	First-in→	0, 1, 2
1	Hit		First-in→	0, 1, 2

Figure 22.2: **Tracing The FIFO Policy**

that was the "first one" in (the cache state in the figure is kept in FIFO order, with the first-in page on the left), which is page 0. Unfortunately, our next access is to page 0, causing another miss and replacement (of page 1). We then hit on page 3, but miss on 1 and 2, and finally hit on 3.

Comparing FIFO to optimal, FIFO does notably worse: a 36.4% hit rate (or 57.1% excluding compulsory misses). FIFO simply can't determine the importance of blocks: even though page 0 had been accessed a number of times, FIFO still kicks it out, simply because it was the first one brought into memory.

ASIDE: **BELADY'S ANOMALY**

Belady (of the optimal policy) and colleagues found an interesting reference stream that behaved a little unexpectedly [BNS69]. The memory-reference stream: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. The replacement policy they were studying was FIFO. The interesting part: how the cache hit rate changed when moving from a cache size of 3 to 4 pages.

In general, you would expect the cache hit rate to *increase* (get better) when the cache gets larger. But in this case, with FIFO, it gets worse! Calculate the hits and misses yourself and see. This odd behavior is generally referred to as **Belady's Anomaly** (to the chagrin of his co-authors).

Some other policies, such as LRU, don't suffer from this problem. Can you guess why? As it turns out, LRU has what is known as a **stack property** [M+70]. For algorithms with this property, a cache of size $N + 1$ naturally includes the contents of a cache of size N. Thus, when increasing the cache size, hit rate will either stay the same or improve. FIFO and Random (among others) clearly do not obey the stack property, and thus are susceptible to anomalous behavior.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	0	1, 2, 3
0	Miss	1	2, 3, 0
3	Hit		2, 3, 0
1	Miss	3	2, 0, 1
2	Hit		2, 0, 1
1	Hit		2, 0, 1

Figure 22.3: **Tracing The Random Policy**

22.4 Another Simple Policy: Random

Another similar replacement policy is Random, which simply picks a random page to replace under memory pressure. Random has properties similar to FIFO; it is simple to implement, but it doesn't really try to be too intelligent in picking which blocks to evict. Let's look at how Random does on our famous example reference stream (see Figure 22.3).

Of course, how Random does depends entirely upon how lucky (or unlucky) Random gets in its choices. In the example above, Random does a little better than FIFO, and a little worse than optimal. In fact, we can run the Random experiment thousands of times and determine how it does in general. Figure 22.4 shows how many hits Random achieves over 10,000 trials, each with a different random seed. As you can see, sometimes (just over 40% of the time), Random is as good as optimal, achieving 6 hits on the example trace; sometimes it does much worse, achieving 2 hits or fewer. How Random does depends on the luck of the draw.

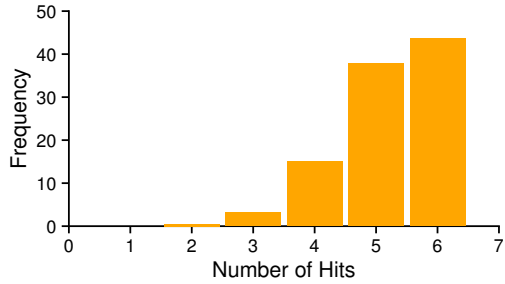


Figure 22.4: **Random Performance Over 10,000 Trials**

Access	Hit/Miss?	Evict	Resulting Cache State	
0	Miss		LRU→	0
1	Miss		LRU→	0, 1
2	Miss		LRU→	0, 1, 2
0	Hit		LRU→	1, 2, 0
1	Hit		LRU→	2, 0, 1
3	Miss	2	LRU→	0, 1, 3
0	Hit		LRU→	1, 3, 0
3	Hit		LRU→	1, 0, 3
1	Hit		LRU→	0, 3, 1
2	Miss	0	LRU→	3, 1, 2
1	Hit		LRU→	3, 2, 1

Figure 22.5: **Tracing The LRU Policy**

22.5 Using History: LRU

Unfortunately, any policy as simple as FIFO or Random is likely to have a common problem: it might kick out an important page, one that is about to be referenced again. FIFO kicks out the page that was first brought in; if this happens to be a page with important code or data structures upon it, it gets thrown out anyhow, even though it will soon be paged back in. Thus, FIFO, Random, and similar policies are not likely to approach optimal; something smarter is needed.

As we did with scheduling policy, to improve our guess at the future, we once again lean on the past and use *history* as our guide. For example, if a program has accessed a page in the near past, it is likely to access it again in the near future.

One type of historical information a page-replacement policy could use is **frequency**; if a page has been accessed many times, perhaps it should not be replaced as it clearly has some value. A more commonly-used property of a page is its **recency** of access; the more recently a page has been accessed, perhaps the more likely it will be accessed again.

This family of policies is based on what people refer to as the **principle of locality** [D70], which basically is just an observation about programs and their behavior. What this principle says, quite simply, is that programs tend to access certain code sequences (e.g., in a loop) and data structures (e.g., an array accessed by the loop) quite frequently; we should thus try to use history to figure out which pages are important, and keep those pages in memory when it comes to eviction time.

And thus, a family of simple historically-based algorithms are born. The **Least-Frequently-Used** (**LFU**) policy replaces the least-frequently-used page when an eviction must take place. Similarly, the **Least-Recently-Used** (**LRU**) policy replaces the least-recently-used page. These algorithms are easy to remember: once you know the name, you know exactly what it does, which is an excellent property for a name.

To better understand LRU, let's examine how LRU does on our exam-

ASIDE: **TYPES OF LOCALITY**

There are two types of locality that programs tend to exhibit. The first is known as **spatial locality**, which states that if a page P is accessed, it is likely the pages around it (say $P-1$ or $P+1$) will also likely be accessed. The second is **temporal locality**, which states that pages that have been accessed in the near past are likely to be accessed again in the near future. The assumption of the presence of these types of locality plays a large role in the caching hierarchies of hardware systems, which deploy many levels of instruction, data, and address-translation caching to help programs run fast when such locality exists.

Of course, the **principle of locality**, as it is often called, is no hard-and-fast rule that all programs must obey. Indeed, some programs access memory (or disk) in rather random fashion and don't exhibit much or any locality in their access streams. Thus, while locality is a good thing to keep in mind while designing caches of any kind (hardware or software), it does not *guarantee* success. Rather, it is a heuristic that often proves useful in the design of computer systems.

ple reference stream. Figure 22.5 (page 7) shows the results. From the figure, you can see how LRU can use history to do better than stateless policies such as Random or FIFO. In the example, LRU evicts page 2 when it first has to replace a page, because 0 and 1 have been accessed more recently. It then replaces page 0 because 1 and 3 have been accessed more recently. In both cases, LRU's decision, based on history, turns out to be correct, and the next references are thus hits. Thus, in our simple example, LRU does as well as possible, matching optimal in its performance[2].

We should also note that the opposites of these algorithms exist: **Most-Frequently-Used** (**MFU**) and **Most-Recently-Used** (**MRU**). In most cases (not all!), these policies do not work well, as they ignore the locality most programs exhibit instead of embracing it.

22.6 Workload Examples

Let's look at a few more examples in order to better understand how some of these policies behave. Here, we'll examine more complex **workloads** instead of small traces. However, even these workloads are greatly simplified; a better study would include application traces.

Our first workload has no locality, which means that each reference is to a random page within the set of accessed pages. In this simple example, the workload accesses 100 unique pages over time, choosing the next page to refer to at random; overall, 10,000 pages are accessed. In the experiment, we vary the cache size from very small (1 page) to enough to hold all the unique pages (100 page), in order to see how each policy behaves over the range of cache sizes.

[2] OK, we cooked the results. But sometimes cooking is necessary to prove a point.

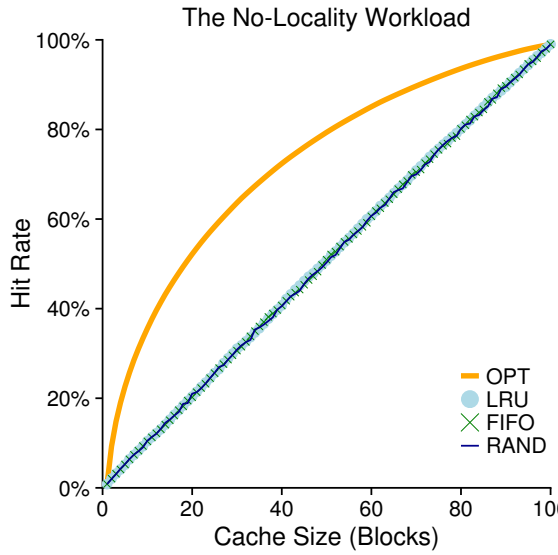


Figure 22.6: **The No-Locality Workload**

Figure 22.6 plots the results of the experiment for optimal, LRU, Random, and FIFO. The y-axis of the figure shows the hit rate that each policy achieves; the x-axis varies the cache size as described above.

We can draw a number of conclusions from the graph. First, when there is no locality in the workload, it doesn't matter much which realistic policy you are using; LRU, FIFO, and Random all perform the same, with the hit rate exactly determined by the size of the cache. Second, when the cache is large enough to fit the entire workload, it also doesn't matter which policy you use; all policies (even Random) converge to a 100% hit rate when all the referenced blocks fit in cache. Finally, you can see that optimal performs noticeably better than the realistic policies; peeking into the future, if it were possible, does a much better job of replacement.

The next workload we examine is called the "80-20" workload, which exhibits locality: 80% of the references are made to 20% of the pages (the "hot" pages); the remaining 20% of the references are made to the remaining 80% of the pages (the "cold" pages). In our workload, there are a total 100 unique pages again; thus, "hot" pages are referred to most of the time, and "cold" pages the remainder. Figure 22.7 (page 10) shows how the policies perform with this workload.

As you can see from the figure, while both random and FIFO do reasonably well, LRU does better, as it is more likely to hold onto the hot pages; as those pages have been referred to frequently in the past, they are likely to be referred to again in the near future. Optimal once again does better, showing that LRU's historical information is not perfect.

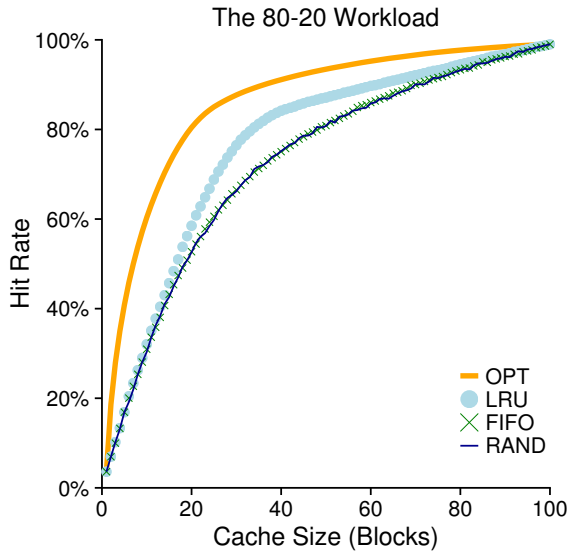


Figure 22.7: **The 80-20 Workload**

You might now be wondering: is LRU's improvement over Random and FIFO really that big of a deal? The answer, as usual, is "it depends." If each miss is very costly (not uncommon), then even a small increase in hit rate (reduction in miss rate) can make a huge difference on performance. If misses are not so costly, then of course the benefits possible with LRU are not nearly as important.

Let's look at one final workload. We call this one the "looping sequential" workload, as in it, we refer to 50 pages in sequence, starting at 0, then 1, ..., up to page 49, and then we loop, repeating those accesses, for a total of 10,000 accesses to 50 unique pages. The last graph in Figure 22.8 shows the behavior of the policies under this workload.

This workload, common in many applications (including important commercial applications such as databases [CD85]), represents a worst-case for both LRU and FIFO. These algorithms, under a looping-sequential workload, kick out older pages; unfortunately, due to the looping nature of the workload, these older pages are going to be accessed sooner than the pages that the policies prefer to keep in cache. Indeed, even with a cache of size 49, a looping-sequential workload of 50 pages results in a 0% hit rate. Interestingly, Random fares notably better, not quite approaching optimal, but at least achieving a non-zero hit rate. Turns out that random has some nice properties; one such property is not having weird corner-case behaviors.

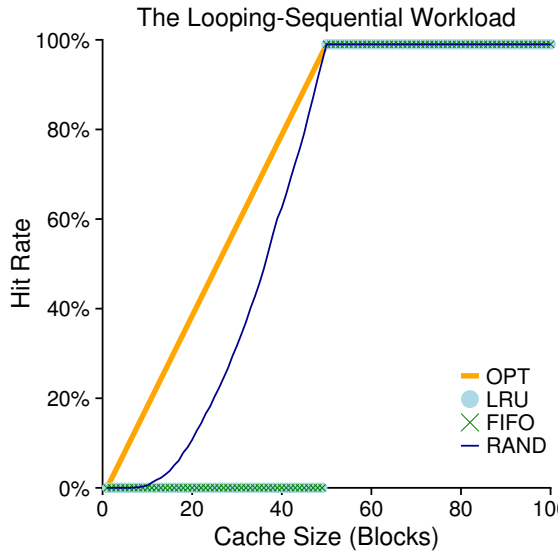


Figure 22.8: **The Looping Workload**

22.7 Implementing Historical Algorithms

As you can see, an algorithm such as LRU can generally do a better job than simpler policies like FIFO or Random, which may throw out important pages. Unfortunately, historical policies present us with a new challenge: how do we implement them?

Let's take, for example, LRU. To implement it perfectly, we need to do a lot of work. Specifically, upon each *page access* (i.e., each memory access, whether an instruction fetch or a load or store), we must update some data structure to move this page to the front of the list (i.e., the MRU side). Contrast this to FIFO, where the FIFO list of pages is only accessed when a page is *evicted* (by removing the first-in page) or when a new page is added to the list (to the last-in side). To keep track of which pages have been least- and most-recently used, the system has to do some accounting work *on every memory reference*. Clearly, without great care, such accounting could greatly reduce performance.

One method that could help speed this up is to add a little bit of hardware support. For example, a machine could update, on each page access, a time field in memory (for example, this could be in the per-process page table, or just in some separate array in memory, with one entry per physical page of the system). Thus, when a page is accessed, the time field would be set, by hardware, to the current time. Then, when replacing a page, the OS could simply scan all the time fields in the system to find the least-recently-used page.

Unfortunately, as the number of pages in a system grows, scanning a huge array of times just to find the absolute least-recently-used page is prohibitively expensive. Imagine a modern machine with 4GB of memory, chopped into 4KB pages. This machine has 1 million pages, and thus finding the LRU page will take a long time, even at modern CPU speeds. Which begs the question: do we really need to find the absolute oldest page to replace? Can we instead survive with an approximation?

CRUX: HOW TO IMPLEMENT AN LRU REPLACEMENT POLICY
Given that it will be expensive to implement perfect LRU, can we approximate it in some way, and still obtain the desired behavior?

22.8 Approximating LRU

As it turns out, the answer is yes: approximating LRU is more feasible from a computational-overhead standpoint, and indeed it is what many modern systems do. The idea requires some hardware support, in the form of a **use bit** (sometimes called the **reference bit**), the first of which was implemented in the first system with paging, the Atlas one-level store [KE+62]. There is one use bit per page of the system, and the use bits live in memory somewhere (they could be in the per-process page tables, for example, or just in an array somewhere). Whenever a page is referenced (i.e., read or written), the use bit is set by hardware to 1. The hardware never clears the bit, though (i.e., sets it to 0); that is the responsibility of the OS.

How does the OS employ the use bit to approximate LRU? Well, there could be a lot of ways, but with the **clock algorithm** [C69], one simple approach was suggested. Imagine all the pages of the system arranged in a circular list. A **clock hand** points to some particular page to begin with (it doesn't really matter which). When a replacement must occur, the OS checks if the currently-pointed to page P has a use bit of 1 or 0. If 1, this implies that page P was recently used and thus is *not* a good candidate for replacement. Thus, the use bit for P set to 0 (cleared), and the clock hand is incremented to the next page ($P + 1$). The algorithm continues until it finds a use bit that is set to 0, implying this page has not been recently used (or, in the worst case, that all pages have been and that we have now searched through the entire set of pages, clearing all the bits).

Note that this approach is not the only way to employ a use bit to approximate LRU. Indeed, any approach which periodically clears the use bits and then differentiates between which pages have use bits of 1 versus 0 to decide which to replace would be fine. The clock algorithm of Corbato's was just one early approach which met with some success, and had the nice property of not repeatedly scanning through all of memory looking for an unused page.

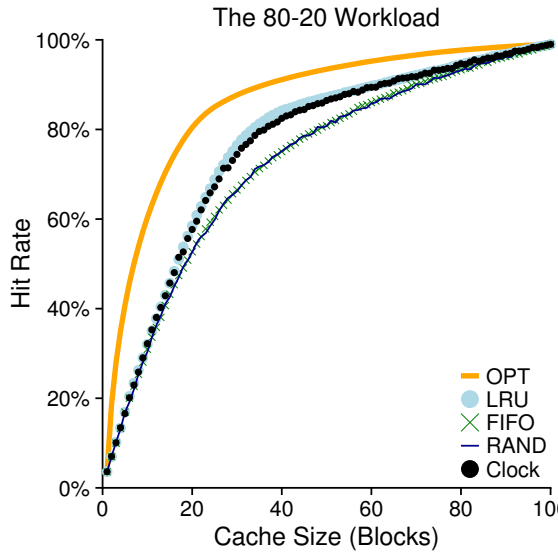


Figure 22.9: **The 80-20 Workload With Clock**

The behavior of a clock algorithm variant is shown in Figure 22.9. This variant randomly scans pages when doing a replacement; when it encounters a page with a reference bit set to 1, it clears the bit (i.e., sets it to 0); when it finds a page with the reference bit set to 0, it chooses it as its victim. As you can see, although it doesn't do quite as well as perfect LRU, it does better than approaches that don't consider history at all.

22.9 Considering Dirty Pages

One small modification to the clock algorithm (also originally suggested by Corbato [C69]) that is commonly made is the additional consideration of whether a page has been modified or not while in memory. The reason for this: if a page has been **modified** and is thus **dirty**, it must be written back to disk to evict it, which is expensive. If it has not been modified (and is thus **clean**), the eviction is free; the physical frame can simply be reused for other purposes without additional I/O. Thus, some VM systems prefer to evict clean pages over dirty pages.

To support this behavior, the hardware should include a **modified bit** (a.k.a. **dirty bit**). This bit is set any time a page is written, and thus can be incorporated into the page-replacement algorithm. The clock algorithm, for example, could be changed to scan for pages that are both unused and clean to evict first; failing to find those, then for unused pages that are dirty, and so forth.

22.10 Other VM Policies

Page replacement is not the only policy the VM subsystem employs (though it may be the most important). For example, the OS also has to decide *when* to bring a page into memory. This policy, sometimes called the **page selection** policy (as it was called by Denning [D70]), presents the OS with some different options.

For most pages, the OS simply uses **demand paging**, which means the OS brings the page into memory when it is accessed, "on demand" as it were. Of course, the OS could guess that a page is about to be used, and thus bring it in ahead of time; this behavior is known as **prefetching** and should only be done when there is reasonable chance of success. For example, some systems will assume that if a code page P is brought into memory, that code page $P+1$ will likely soon be accessed and thus should be brought into memory too.

Another policy determines how the OS writes pages out to disk. Of course, they could simply be written out one at a time; however, many systems instead collect a number of pending writes together in memory and write them to disk in one (more efficient) write. This behavior is usually called **clustering** or simply **grouping** of writes, and is effective because of the nature of disk drives, which perform a single large write more efficiently than many small ones.

22.11 Thrashing

Before closing, we address one final question: what should the OS do when memory is simply oversubscribed, and the memory demands of the set of running processes simply exceeds the available physical memory? In this case, the system will constantly be paging, a condition sometimes referred to as **thrashing** [D70].

Some earlier operating systems had a fairly sophisticated set of mechanisms to both detect and cope with thrashing when it took place. For example, given a set of processes, a system could decide not to run a subset of processes, with the hope that the reduced set of processes' **working sets** (the pages that they are using actively) fit in memory and thus can make progress. This approach, generally known as **admission control**, states that it is sometimes better to do less work well than to try to do everything at once poorly, a situation we often encounter in real life as well as in modern computer systems (sadly).

Some current systems take more a draconian approach to memory overload. For example, some versions of Linux run an **out-of-memory killer** when memory is oversubscribed; this daemon chooses a memory-intensive process and kills it, thus reducing memory in a none-too-subtle manner. While successful at reducing memory pressure, this approach can have problems, if, for example, it kills the X server and thus renders any applications requiring the display unusable.

22.12 Summary

We have seen the introduction of a number of page-replacement (and other) policies, which are part of the VM subsystem of all modern operating systems. Modern systems add some tweaks to straightforward LRU approximations like clock; for example, **scan resistance** is an important part of many modern algorithms, such as ARC [MM03]. Scan-resistant algorithms are usually LRU-like but also try to avoid the worst-case behavior of LRU, which we saw with the looping-sequential workload. Thus, the evolution of page-replacement algorithms continues.

However, in many cases the importance of said algorithms has decreased, as the discrepancy between memory-access and disk-access times has increased. Because paging to disk is so expensive, the cost of frequent paging is prohibitive. Thus, the best solution to excessive paging is often a simple (if intellectually dissatisfying) one: buy more memory.

References

[AD03] "Run-Time Adaptation in River"
Remzi H. Arpaci-Dusseau
ACM TOCS, 21:1, February 2003
A summary of one of the authors' dissertation work on a system named River. Certainly one place where he learned that comparison against the ideal is an important technique for system designers.

[B66] "A Study of Replacement Algorithms for Virtual-Storage Computer"
Laszlo A. Belady
IBM Systems Journal 5(2): 78-101, 1966
The paper that introduces the simple way to compute the optimal behavior of a policy (the MIN algorithm).

[BNS69] "An Anomaly in Space-time Characteristics of Certain Programs Running in a Paging Machine"
L. A. Belady and R. A. Nelson and G. S. Shedler
Communications of the ACM, 12:6, June 1969
Introduction of the little sequence of memory references known as Belady's Anomaly. How do Nelson and Shedler feel about this name, we wonder?

[CD85] "An Evaluation of Buffer Management Strategies for Relational Database Systems"
Hong-Tai Chou and David J. DeWitt
VLDB '85, Stockholm, Sweden, August 1985
A famous database paper on the different buffering strategies you should use under a number of common database access patterns. The more general lesson: if you know something about a workload, you can tailor policies to do better than the general-purpose ones usually found in the OS.

[C69] "A Paging Experiment with the Multics System"
F.J. Corbato
Included in a Festschrift published in honor of Prof. P.M. Morse
MIT Press, Cambridge, MA, 1969
The original (and hard to find!) reference to the clock algorithm, though not the first usage of a use bit. Thanks to H. Balakrishnan of MIT for digging up this paper for us.

[D70] "Virtual Memory"
Peter J. Denning
Computing Surveys, Vol. 2, No. 3, September 1970
Denning's early and famous survey on virtual memory systems.

[EF78] "Cold-start vs. Warm-start Miss Ratios"
Malcolm C. Easton and Ronald Fagin
Communications of the ACM, 21:10, October 1978
A good discussion of cold-start vs. warm-start misses.

[FP89] "Electrochemically Induced Nuclear Fusion of Deuterium"
Martin Fleischmann and Stanley Pons
Journal of Electroanalytical Chemistry, Volume 26, Number 2, Part 1, April, 1989
The famous paper that would have revolutionized the world in providing an easy way to generate nearly-infinite power from jars of water with a little metal in them. Unforuntately, the results published (and widely publicized) by Pons and Fleischmann turned out to be impossible to reproduce, and thus these two well-meaning scientists were discredited (and certainly, mocked). The only guy really happy about this result was Marvin Hawkins, whose name was left off this paper even though he participated in the work; he thus avoided having his name associated with one of the biggest scientific goofs of the 20th century.

[HP06] "Computer Architecture: A Quantitative Approach"
John Hennessy and David Patterson
Morgan-Kaufmann, 2006
A great and marvelous book about computer architecture. Read it!

[H87] "Aspects of Cache Memory and Instruction Buffer Performance"
Mark D. Hill
Ph.D. Dissertation, U.C. Berkeley, 1987
Mark Hill, in his dissertation work, introduced the Three C's, which later gained wide popularity with its inclusion in H&P [HP06]. The quote from therein: "I have found it useful to partition misses ... into three components intuitively based on the cause of the misses (page 49)."

[KE+62] "One-level Storage System"
T. Kilburn, and D.B.G. Edwards and M.J. Lanigan and F.H. Sumner
IRE Trans. EC-11:2, 1962
Although Atlas had a use bit, it only had a very small number of pages, and thus the scanning of the use bits in large memories was not a problem the authors solved.

[M+70] "Evaluation Techniques for Storage Hierarchies"
R. L. Mattson, J. Gecsei, D. R. Slutz, I. L. Traiger
IBM Systems Journal, Volume 9:2, 1970
A paper that is mostly about how to simulate cache hierarchies efficiently; certainly a classic in that regard, as well for its excellent discussion of some of the properties of various replacement algorithms. Can you figure out why the stack property might be useful for simulating a lot of different-sized caches at once?

[MM03] "ARC: A Self-Tuning, Low Overhead Replacement Cache"
Nimrod Megiddo and Dharmendra S. Modha
FAST 2003, February 2003, San Jose, California
An excellent modern paper about replacement algorithms, which includes a new policy, ARC, that is now used in some systems. Recognized in 2014 as a "Test of Time" award winner by the storage systems community at the FAST '14 conference.

Homework

This simulator, `paging-policy.py`, allows you to play around with different page-replacement policies. See the README for details.

Questions

1. Generate random addresses with the following arguments: `-s 0 -n 10`, `-s 1 -n 10`, and `-s 2 -n 10`. Change the policy from FIFO, to LRU, to OPT. Compute whether each access in said address traces are hits or misses.
2. For a cache of size 5, generate worst-case address reference streams for each of the following policies: FIFO, LRU, and MRU (worst-case reference streams cause the most misses possible. For the worst case reference streams, how much bigger of a cache is needed to improve performance dramatically and approach OPT?
3. Generate a random trace (use python or perl). How would you expect the different policies to perform on such a trace?
4. Now generate a trace with some locality. How can you generate such a trace? How does LRU perform on it? How much better than RAND is LRU? How does CLOCK do? How about CLOCK with different numbers of clock bits?
5. Use a program like `valgrind` to instrument a real application and generate a virtual page reference stream. For example, running `valgrind --tool=lackey --trace-mem=yes ls` will output a nearly-complete reference trace of every instruction and data reference made by the program `ls`. To make this useful for the simulator above, you'll have to first transform each virtual memory reference into a virtual page-number reference (done by masking off the offset and shifting the resulting bits downward). How big of a cache is needed for your application trace in order to satisfy a large fraction of requests? Plot a graph of its working set as the size of the cache increases.

23

The VAX/VMS Virtual Memory System

Before we end our study of virtual memory, let us take a closer look at one particularly clean and well done virtual memory manager, that found in the VAX/VMS operating system [LL82]. In this note, we will discuss the system to illustrate how some of the concepts brought forth in earlier chapters together in a complete memory manager.

23.1 Background

The VAX-11 minicomputer architecture was introduced in the late 1970's by **Digital Equipment Corporation** (**DEC**). DEC was a massive player in the computer industry during the era of the mini-computer; unfortunately, a series of bad decisions and the advent of the PC slowly (but surely) led to their demise [C03]. The architecture was realized in a number of implementations, including the VAX-11/780 and the less powerful VAX-11/750.

The OS for the system was known as VAX/VMS (or just plain VMS), one of whose primary architects was Dave Cutler, who later led the effort to develop Microsoft's Windows NT [C93]. VMS had the general problem that it would be run on a broad range of machines, including very inexpensive VAXen (yes, that is the proper plural) to extremely high-end and powerful machines in the same architecture family. Thus, the OS had to have mechanisms and policies that worked (and worked well) across this huge range of systems.

THE CRUX: HOW TO AVOID THE CURSE OF GENERALITY

Operating systems often have a problem known as "the curse of generality", where they are tasked with general support for a broad class of applications and systems. The fundamental result of the curse is that the OS is not likely to support any one installation very well. In the case of VMS, the curse was very real, as the VAX-11 architecture was realized in a number of different implementations. Thus, how can an OS be built so as to run effectively on a wide range of systems?

As an additional issue, VMS is an excellent example of software innovations used to hide some of the inherent flaws of the architecture. Although the OS often relies on the hardware to build efficient abstractions and illusions, sometimes the hardware designers don't quite get everything right; in the VAX hardware, we'll see a few examples of this, and what the VMS operating system does to build an effective, working system despite these hardware flaws.

23.2 Memory Management Hardware

The VAX-11 provided a 32-bit virtual address space per process, divided into 512-byte pages. Thus, a virtual address consisted of a 23-bit VPN and a 9-bit offset. Further, the upper two bits of the VPN were used to differentiate which segment the page resided within; thus, the system was a hybrid of paging and segmentation, as we saw previously.

The lower-half of the address space was known as "process space" and is unique to each process. In the first half of process space (known as `P0`), the user program is found, as well as a heap which grows downward. In the second half of process space (`P1`), we find the stack, which grows upwards. The upper-half of the address space is known as system space (`S`), although only half of it is used. Protected OS code and data reside here, and the OS is in this way shared across processes.

One major concern of the VMS designers was the incredibly small size of pages in the VAX hardware (512 bytes). This size, chosen for historical reasons, has the fundamental problem of making simple linear page tables excessively large. Thus, one of the first goals of the VMS designers was to make sure that VMS would not overwhelm memory with page tables.

The system reduced the pressure page tables place on memory in two ways. First, by segmenting the user address space into two, the VAX-11 provides a page table for each of these regions (`P0` and `P1`) per process; thus, no page-table space is needed for the unused portion of the address space between the stack and the heap. The base and bounds registers are used as you would expect; a base register holds the address of the page table for that segment, and the bounds holds its size (i.e., number of page-table entries).

Second, the OS reduces memory pressure even further by placing user page tables (for `P0` and `P1`, thus two per process) in kernel virtual memory. Thus, when allocating or growing a page table, the kernel allocates space out of its own virtual memory, in segment `S`. If memory comes under severe pressure, the kernel can swap pages of these page tables out to disk, thus making physical memory available for other uses.

Putting page tables in kernel virtual memory means that address translation is even further complicated. For example, to translate a virtual address in `P0` or `P1`, the hardware has to first try to look up the page-table entry for that page in its page table (the `P0` or `P1` page table for that pro-

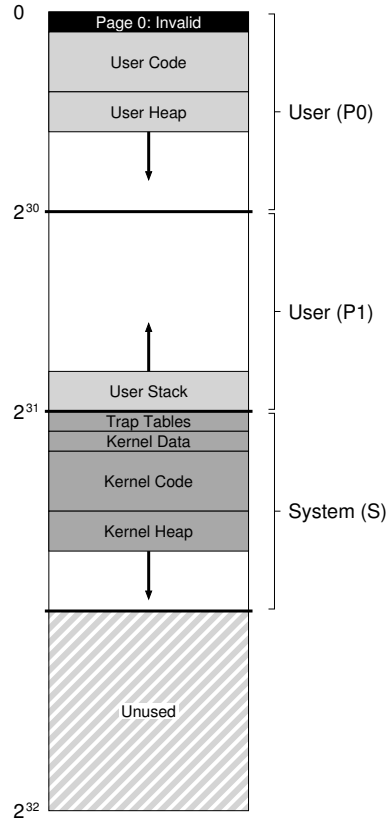


Figure 23.1: **The VAX/VMS Address Space**

cess); in doing so, however, the hardware may first have to consult the system page table (which lives in physical memory); with that translation complete, the hardware can learn the address of the page of the page table, and then finally learn the address of the desired memory access. All of this, fortunately, is made faster by the VAX's hardware-managed TLBs, which usually (hopefully) circumvent this laborious lookup.

23.3 A Real Address Space

One neat aspect of studying VMS is that we can see how a real address space is constructed (Figure 23.1. Thus far, we have assumed a simple address space of just user code, user data, and user heap, but as we can see above, a real address space is notably more complex.

ASIDE: **WHY NULL POINTER ACCESSES CAUSE SEG FAULTS**
You should now have a good understanding of exactly what happens on a null-pointer dereference. A process generates a virtual address of 0, by doing something like this:

```
int *p = NULL; // set p = 0
*p = 10;       // try to store value 10 to virtual address 0
```

The hardware tries to look up the VPN (also 0 here) in the TLB, and suffers a TLB miss. The page table is consulted, and the entry for VPN 0 is found to be marked invalid. Thus, we have an invalid access, which transfers control to the OS, which likely terminates the process (on UNIX systems, processes are sent a signal which allows them to react to such a fault; if uncaught, however, the process is killed).

For example, the code segment never begins at page 0. This page, instead, is marked inaccessible, in order to provide some support for detecting **null-pointer** accesses. Thus, one concern when designing an address space is support for debugging, which the inaccessible zero page provides here in some form.

Perhaps more importantly, the kernel virtual address space (i.e., its data structures and code) is a part of each user address space. On a context switch, the OS changes the `P0` and `P1` registers to point to the appropriate page tables of the soon-to-be-run process; however, it does not change the `S` base and bound registers, and as a result the "same" kernel structures are mapped into each user address space.

The kernel is mapped into each address space for a number of reasons. This construction makes life easier for the kernel; when, for example, the OS is handed a pointer from a user program (e.g., on a `write()` system call), it is easy to copy data from that pointer to its own structures. The OS is naturally written and compiled, without worry of where the data it is accessing comes from. If in contrast the kernel were located entirely in physical memory, it would be quite hard to do things like swap pages of the page table to disk; if the kernel were given its own address space, moving data between user applications and the kernel would again be complicated and painful. With this construction (now used widely), the kernel appears almost as a library to applications, albeit a protected one.

One last point about this address space relates to protection. Clearly, the OS does not want user applications reading or writing OS data or code. Thus, the hardware must support different protection levels for pages to enable this. The VAX did so by specifying, in protection bits in the page table, what privilege level the CPU must be at in order to access a particular page. Thus, system data and code are set to a higher level of protection than user data and code; an attempted access to such information from user code will generate a trap into the OS, and (you guessed it) the likely termination of the offending process.

23.4 Page Replacement

The page table entry (PTE) in VAX contains the following bits: a valid bit, a protection field (4 bits), a modify (or dirty) bit, a field reserved for OS use (5 bits), and finally a physical frame number (PFN) to store the location of the page in physical memory. The astute reader might note: no **reference bit**! Thus, the VMS replacement algorithm must make do without hardware support for determining which pages are active.

The developers were also concerned about **memory hogs**, programs that use a lot of memory and make it hard for other programs to run. Most of the policies we have looked at thus far are susceptible to such hogging; for example, LRU is a *global* policy that doesn't share memory fairly among processes.

Segmented FIFO

To address these two problems, the developers came up with the **segmented FIFO** replacement policy [RL81]. The idea is simple: each process has a maximum number of pages it can keep in memory, known as its **resident set size** (**RSS**). Each of these pages is kept on a FIFO list; when a process exceeds its RSS, the "first-in" page is evicted. FIFO clearly does not need any support from the hardware, and is thus easy to implement.

Of course, pure FIFO does not perform particularly well, as we saw earlier. To improve FIFO's performance, VMS introduced two **second-chance lists** where pages are placed before getting evicted from memory, specifically a global *clean-page free list* and *dirty-page list*. When a process P exceeds its RSS, a page is removed from its per-process FIFO; if clean (not modified), it is placed on the end of the clean-page list; if dirty (modified), it is placed on the end of the dirty-page list.

If another process Q needs a free page, it takes the first free page off of the global clean list. However, if the original process P faults on that page *before* it is reclaimed, P reclaims it from the free (or dirty) list, thus avoiding a costly disk access. The bigger these global second-chance lists are, the closer the segmented FIFO algorithm performs to LRU [RL81].

Page Clustering

Another optimization used in VMS also helps overcome the small page size in VMS. Specifically, with such small pages, disk I/O during swapping could be highly inefficient, as disks do better with large transfers. To make swapping I/O more efficient, VMS adds a number of optimizations, but most important is **clustering**. With clustering, VMS groups large batches of pages together from the global dirty list, and writes them to disk in one fell swoop (thus making them clean). Clustering is used in most modern systems, as the freedom to place pages anywhere within swap space lets the OS group pages, perform fewer and bigger writes, and thus improve performance.

ASIDE: **EMULATING REFERENCE BITS**

As it turns out, you don't need a hardware reference bit in order to get some notion of which pages are in use in a system. In fact, in the early 1980's, Babaoglu and Joy showed that protection bits on the VAX can be used to emulate reference bits [BJ81]. The basic idea: if you want to gain some understanding of which pages are actively being used in a system, mark all of the pages in the page table as inaccessible (but keep around the information as to which pages are really accessible by the process, perhaps in the "reserved OS field" portion of the page table entry). When a process accesses a page, it will generate a trap into the OS; the OS will then check if the page really should be accessible, and if so, revert the page to its normal protections (e.g., read-only, or read-write). At the time of a replacement, the OS can check which pages remain marked inaccessible, and thus get an idea of which pages have not been recently used.

The key to this "emulation" of reference bits is reducing overhead while still obtaining a good idea of page usage. The OS must not be too aggressive in marking pages inaccessible, or overhead would be too high. The OS also must not be too passive in such marking, or all pages will end up referenced; the OS will again have no good idea which page to evict.

23.5 Other Neat VM Tricks

VMS had two other now-standard tricks: demand zeroing and copy-on-write. We now describe these **lazy** optimizations.

One form of laziness in VMS (and most modern systems) is **demand zeroing** of pages. To understand this better, let's consider the example of adding a page to your address space, say in your heap. In a naive implementation, the OS responds to a request to add a page to your heap by finding a page in physical memory, zeroing it (required for security; otherwise you'd be able to see what was on the page from when some other process used it!), and then mapping it into your address space (i.e., setting up the page table to refer to that physical page as desired). But the naive implementation can be costly, particularly if the page does not get used by the process.

With demand zeroing, the OS instead does very little work when the page is added to your address space; it puts an entry in the page table that marks the page inaccessible. If the process then reads or writes the page, a trap into the OS takes place. When handling the trap, the OS notices (usually through some bits marked in the "reserved for OS" portion of the page table entry) that this is actually a demand-zero page; at this point, the OS then does the needed work of finding a physical page, zeroing it, and mapping it into the process's address space. If the process never accesses the page, all of this work is avoided, and thus the virtue of demand zeroing.

TIP: BE LAZY

Being lazy can be a virtue in both life as well as in operating systems. Laziness can put off work until later, which is beneficial within an OS for a number of reasons. First, putting off work might reduce the latency of the current operation, thus improving responsiveness; for example, operating systems often report that writes to a file succeeded immediately, and only write them to disk later in the background. Second, and more importantly, laziness sometimes obviates the need to do the work at all; for example, delaying a write until the file is deleted removes the need to do the write at all. Laziness is also good in life: for example, by putting off your OS project, you may find that the project specification bugs are worked out by your fellow classmates; however, the class project is unlikely to get canceled, so being too lazy may be problematic, leading to a late project, bad grade, and a sad professor. Don't make professors sad!

Another cool optimization found in VMS (and again, in virtually every modern OS) is **copy-on-write** (**COW** for short). The idea, which goes at least back to the TENEX operating system [BB+72], is simple: when the OS needs to copy a page from one address space to another, instead of copying it, it can map it into the target address space and mark it read-only in both address spaces. If both address spaces only read the page, no further action is taken, and thus the OS has realized a fast copy without actually moving any data.

If, however, one of the address spaces does indeed try to write to the page, it will trap into the OS. The OS will then notice that the page is a COW page, and thus (lazily) allocate a new page, fill it with the data, and map this new page into the address space of the faulting process. The process then continues and now has its own private copy of the page.

COW is useful for a number of reasons. Certainly any sort of shared library can be mapped copy-on-write into the address spaces of many processes, saving valuable memory space. In UNIX systems, COW is even more critical, due to the semantics of `fork()` and `exec()`. As you might recall, `fork()` creates an exact copy of the address space of the caller; with a large address space, making such a copy is slow and data intensive. Even worse, most of the address space is immediately over-written by a subsequent call to `exec()`, which overlays the calling process's address space with that of the soon-to-be-exec'd program. By instead performing a copy-on-write `fork()`, the OS avoids much of the needless copying and thus retains the correct semantics while improving performance.

23.6 Summary

You have now seen a top-to-bottom review of an entire virtual memory system. Hopefully, most of the details were easy to follow, as you should have already had a good understanding of most of the basic mechanisms and policies. More detail is available in the excellent (and short) paper by Levy and Lipman [LL82]; we encourage you to read it, a great way to see what the source material behind these chapters is like.

You should also learn more about the state of the art by reading about Linux and other modern systems when possible. There is a lot of source material out there, including some reasonable books [BC05]. One thing that will amaze you: how classic ideas, found in old papers such as this one on VAX/VMS, still influence how modern operating systems are built.

References

[BB+72] "TENEX, A Paged Time Sharing System for the PDP-10"
Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, Raymond S. Tomlinson
Communications of the ACM, Volume 15, March 1972
An early time-sharing OS where a number of good ideas came from. Copy-on-write was just one of those; inspiration for many other aspects of modern systems, including process management, virtual memory, and file systems are found herein.

[BJ81] "Converting a Swap-Based System to do Paging
in an Architecture Lacking Page-Reference Bits"
Ozalp Babaoglu and William N. Joy
SOSP '81, Pacific Grove, California, December 1981
A clever idea paper on how to exploit existing protection machinery within a machine in order to emulate reference bits. The idea came from the group at Berkeley working on their own version of UNIX*, known as the Berkeley Systems Distribution, or BSD. The group was heavily influential in the development of* UNIX*, in virtual memory, file systems, and networking.*

[BC05] "Understanding the Linux Kernel (Third Edition)"
Daniel P. Bovet and Marco Cesati
O'Reilly Media, November 2005
One of the many books you can find on Linux. They go out of date quickly, but many of the basics remain and are worth reading about.

[C03] "The Innovator's Dilemma"
Clayton M. Christenson
Harper Paperbacks, January 2003
A fantastic book about the disk-drive industry and how new innovations disrupt existing ones. A good read for business majors and computer scientists alike. Provides insight on how large and successful companies completely fail.

[C93] "Inside Windows NT"
Helen Custer and David Solomon
Microsoft Press, 1993
The book about Windows NT that explains the system top to bottom, in more detail than you might like. But seriously, a pretty good book.

[LL82] "Virtual Memory Management in the VAX/VMS Operating System"
Henry M. Levy, Peter H. Lipman
IEEE Computer, Volume 15, Number 3 (March 1982) *Read the original source of most of this material; it is a concise and easy read. Particularly important if you wish to go to graduate school, where all you do is read papers, work, read some more papers, work more, eventually write a paper, and then work some more. But it is fun!*

[RL81] "Segmented FIFO Page Replacement"
Rollins Turner and Henry Levy
SIGMETRICS '81, Las Vegas, Nevada, September 1981
A short paper that shows for some workloads, segmented FIFO can approach the performance of LRU.

12

A Dialogue on Memory Virtualization

Student: *So, are we done with virtualization?*

Professor: *No!*

Student: *Hey, no reason to get so excited; I was just asking a question. Students are supposed to do that, right?*

Professor: *Well, professors do always say that, but really they mean this: ask questions, **if** they are good questions, **and** you have actually put a little thought into them.*

Student: *Well, that sure takes the wind out of my sails.*

Professor: *Mission accomplished. In any case, we are not nearly done with virtualization! Rather, you have just seen how to virtualize the CPU, but really there is a big monster waiting in the closet: memory. Virtualizing memory is complicated and requires us to understand many more intricate details about how the hardware and OS interact.*

Student: *That sounds cool. Why is it so hard?*

Professor: *Well, there are a lot of details, and you have to keep them straight in your head to really develop a mental model of what is going on. We'll start simple, with very basic techniques like base/bounds, and slowly add complexity to tackle new challenges, including fun topics like TLBs and multi-level page tables. Eventually, we'll be able to describe the workings of a fully-functional modern virtual memory manager.*

Student: *Neat! Any tips for the poor student, inundated with all of this information and generally sleep-deprived?*

Professor: *For the sleep deprivation, that's easy: sleep more (and party less). For understanding virtual memory, start with this: **every address generated by a user program is a virtual address**. The OS is just providing an illusion to each process, specifically that it has its own large and private memory; with some hardware help, the OS will turn these pretend virtual addresses into real physical addresses, and thus be able to locate the desired information.*

Student: *OK, I think I can remember that... (to self) every address from a user program is virtual, every address from a user program is virtual, every ...*

Professor: *What are you mumbling about?*

Student: *Oh nothing.... (awkward pause) ... Anyway, why does the OS want to provide this illusion again?*

Professor: *Mostly **ease of use**: the OS will give each program the view that it has a large contiguous **address space** to put its code and data into; thus, as a programmer, you never have to worry about things like "where should I store this variable?" because the virtual address space of the program is large and has lots of room for that sort of thing. Life, for a programmer, becomes much more tricky if you have to worry about fitting all of your code data into a small, crowded memory.*

Student: *Why else?*

Professor: *Well, **isolation** and **protection** are big deals, too. We don't want one errant program to be able to read, or worse, overwrite, some other program's memory, do we?*

Student: *Probably not. Unless it's a program written by someone you don't like.*

Professor: *Hmmm.... I think we might need to add a class on morals and ethics to your schedule for next semester. Perhaps OS class isn't getting the right message across.*

Student: *Maybe we should. But remember, it's not me who taught us that the proper OS response to errant process behavior is to kill the offending process!*

Part II

Concurrency

25

A Dialogue on Concurrency

Professor: *And thus we reach the second of our three pillars of operating systems:* ***concurrency****.*

Student: *I thought there were four pillars...?*

Professor: *Nope, that was in an older version of the book.*

Student: *Umm... OK. So what is concurrency, oh wonderful professor?*

Professor: *Well, imagine we have a peach —*

Student: *(interrupting) Peaches again! What is it with you and peaches?*

Professor: *Ever read T.S. Eliot? The Love Song of J. Alfred Prufrock, "Do I dare to eat a peach", and all that fun stuff?*

Student: *Oh yes! In English class in high school. Great stuff! I really liked the part where —*

Professor: *(interrupting) This has nothing to do with that — I just like peaches. Anyhow, imagine there are a lot of peaches on a table, and a lot of people who wish to eat them. Let's say we did it this way: each eater first identifies a peach visually, and then tries to grab it and eat it. What is wrong with this approach?*

Student: *Hmmm... seems like you might see a peach that somebody else also sees. If they get there first, when you reach out, no peach for you!*

Professor: *Exactly! So what should we do about it?*

Student: *Well, probably develop a better way of going about this. Maybe form a line, and when you get to the front, grab a peach and get on with it.*

Professor: *Good! But what's wrong with your approach?*

Student: *Sheesh, do I have to do all the work?*

Professor: *Yes.*

Student: *OK, let me think. Well, we used to have many people grabbing for peaches all at once, which is faster. But in my way, we just go one at a time, which is correct, but quite a bit slower. The best kind of approach would be fast and correct, probably.*

Professor: *You are really starting to impress. In fact, you just told us everything we need to know about concurrency! Well done.*

Student: *I did? I thought we were just talking about peaches. Remember, this is usually a part where you make it about computers again.*

Professor: *Indeed. My apologies! One must never forget the concrete. Well, as it turns out, there are certain types of programs that we call* ***multi-threaded*** *applications; each* ***thread*** *is kind of like an independent agent running around in this program, doing things on the program's behalf. But these threads access memory, and for them, each spot of memory is kind of like one of those peaches. If we don't coordinate access to memory between threads, the program won't work as expected. Make sense?*

Student: *Kind of. But why do we talk about this in an OS class? Isn't that just application programming?*

Professor: *Good question! A few reasons, actually. First, the OS must support multi-threaded applications with primitives such as* ***locks*** *and* ***condition variables****, which we'll talk about soon. Second, the OS itself was the first concurrent program — it must access its own memory very carefully or many strange and terrible things will happen. Really, it can get quite grisly.*

Student: *I see. Sounds interesting. There are more details, I imagine?*

Professor: *Indeed there are...*

26

Concurrency: An Introduction

Thus far, we have seen the development of the basic abstractions that the OS performs. We have seen how to take a single physical CPU and turn it into multiple **virtual CPUs**, thus enabling the illusion of multiple programs running at the same time. We have also seen how to create the illusion of a large, private **virtual memory** for each process; this abstraction of the **address space** enables each program to behave as if it has its own memory when indeed the OS is secretly multiplexing address spaces across physical memory (and sometimes, disk).

In this note, we introduce a new abstraction for a single running process: that of a **thread**. Instead of our classic view of a single point of execution within a program (i.e., a single PC where instructions are being fetched from and executed), a **multi-threaded** program has more than one point of execution (i.e., multiple PCs, each of which is being fetched and executed from). Perhaps another way to think of this is that each thread is very much like a separate process, except for one difference: they *share* the same address space and thus can access the same data.

The state of a single thread is thus very similar to that of a process. It has a program counter (PC) that tracks where the program is fetching instructions from. Each thread has its own private set of registers it uses for computation; thus, if there are two threads that are running on a single processor, when switching from running one (T1) to running the other (T2), a **context switch** must take place. The context switch between threads is quite similar to the context switch between processes, as the register state of T1 must be saved and the register state of T2 restored before running T2. With processes, we saved state to a **process control block (PCB)**; now, we'll need one or more **thread control blocks (TCBs)** to store the state of each thread of a process. There is one major difference, though, in the context switch we perform between threads as compared to processes: the address space remains the same (i.e., there is no need to switch which page table we are using).

One other major difference between threads and processes concerns the stack. In our simple model of the address space of a classic process (which we can now call a **single-threaded** process), there is a single stack, usually residing at the bottom of the address space (Figure 26.1, left).

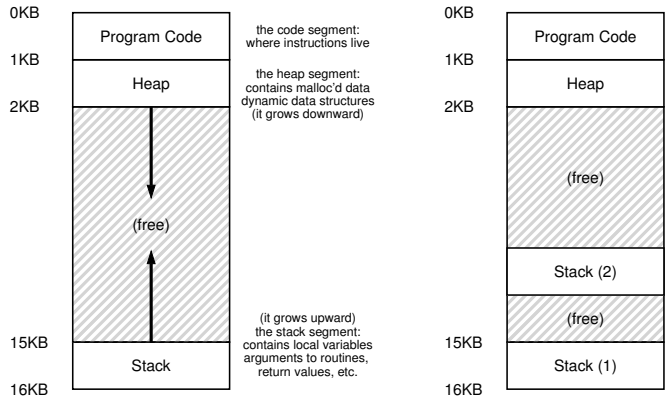


Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

However, in a multi-threaded process, each thread runs independently and of course may call into various routines to do whatever work it is doing. Instead of a single stack in the address space, there will be one per thread. Let's say we have a multi-threaded process that has two threads in it; the resulting address space looks different (Figure 26.1, right).

In this figure, you can see two stacks spread throughout the address space of the process. Thus, any stack-allocated variables, parameters, return values, and other things that we put on the stack will be placed in what is sometimes called **thread-local** storage, i.e., the stack of the relevant thread.

You might also notice how this ruins our beautiful address space layout. Before, the stack and heap could grow independently and trouble only arose when you ran out of room in the address space. Here, we no longer have such a nice situation. Fortunately, this is usually OK, as stacks do not generally have to be very large (the exception being in programs that make heavy use of recursion).

26.1 Why Use Threads?

Before getting into the details of threads and some of the problems you might have in writing multi-threaded programs, let's first answer a more simple question. Why should you use threads at all?

As it turns out, there are at least two major reasons you should use threads. The first is simple: **parallelism**. Imagine you are writing a program that performs operations on very large arrays, for example, adding two large arrays together, or incrementing the value of each element in the array by some amount. If you are running on just a single processor, the task is straightforward: just perform each operation and be done. However, if you are executing the program on a system with multiple

processors, you have the potential of speeding up this process considerably by using the processors to each perform a portion of the work. The task of transforming your standard **single-threaded** program into a program that does this sort of work on multiple CPUs is called **parallelization**, and using a thread per CPU to do this work is a natural and typical way to make programs run faster on modern hardware.

The second reason is a bit more subtle: to avoid blocking program progress due to slow I/O. Imagine that you are writing a program that performs different types of I/O: either waiting to send or receive a message, for an explicit disk I/O to complete, or even (implicitly) for a page fault to finish. Instead of waiting, your program may wish to do something else, including utilizing the CPU to perform computation, or even issuing further I/O requests. Using threads is a natural way to avoid getting stuck; while one thread in your program waits (i.e., is blocked waiting for I/O), the CPU scheduler can switch to other threads, which are ready to run and do something useful. Threading enables **overlap** of I/O with other activities *within* a single program, much like **multiprogramming** did for processes *across* programs; as a result, many modern server-based applications (web servers, database management systems, and the like) make use of threads in their implementations.

Of course, in either of the cases mentioned above, you could use multiple *processes* instead of threads. However, threads share an address space and thus make it easy to share data, and hence are a natural choice when constructing these types of programs. Processes are a more sound choice for logically separate tasks where little sharing of data structures in memory is needed.

26.2 An Example: Thread Creation

Let's get into some of the details. Say we wanted to run a program that creates two threads, each of which does some independent work, in this case printing "A" or "B". The code is shown in Figure 26.2 (page 4).

The main program creates two threads, each of which will run the function `mythread()`, though with different arguments (the string `A` or `B`). Once a thread is created, it may start running right away (depending on the whims of the scheduler); alternately, it may be put in a "ready" but not "running" state and thus not run yet. Of course, on a multiprocessor, the threads could even be running at the same time, but let's not worry about this possibility quite yet.

After creating the two threads (let's call them T1 and T2), the main thread calls `pthread_join()`, which waits for a particular thread to complete. It does so twice, thus ensuring T1 and T2 will run and complete before finally allowing the main thread to run again; when it does, it will print "main: end" and exit. Overall, three threads were employed during this run: the main thread, T1, and T2.

```
1  #include <stdio.h>
2  #include <assert.h>
3  #include <pthread.h>
4
5  void *mythread(void *arg) {
6      printf("%s\n", (char *) arg);
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     pthread_t p1, p2;
13     int rc;
14     printf("main: begin\n");
15     rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
16     rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
17     // join waits for the threads to finish
18     rc = pthread_join(p1, NULL); assert(rc == 0);
19     rc = pthread_join(p2, NULL); assert(rc == 0);
20     printf("main: end\n");
21     return 0;
22 }
```

Figure 26.2: **Simple Thread Creation Code (t0.c)**

Let us examine the possible execution ordering of this little program. In the execution diagram (Figure 26.3, page 5), time increases in the downwards direction, and each column shows when a different thread (the main one, or Thread 1, or Thread 2) is running.

Note, however, that this ordering is not the only possible ordering. In fact, given a sequence of instructions, there are quite a few, depending on which thread the scheduler decides to run at a given point. For example, once a thread is created, it may run immediately, which would lead to the execution shown in Figure 26.4 (page 5).

We also could even see "B" printed before "A", if, say, the scheduler decided to run Thread 2 first even though Thread 1 was created earlier; there is no reason to assume that a thread that is created first will run first. Figure 26.5 (page 5) shows this final execution ordering, with Thread 2 getting to strut its stuff before Thread 1.

As you might be able to see, one way to think about thread creation is that it is a bit like making a function call; however, instead of first executing the function and then returning to the caller, the system instead creates a new thread of execution for the routine that is being called, and it runs independently of the caller, perhaps before returning from the create, but perhaps much later. What runs next is determined by the OS **scheduler**, and although the scheduler likely implements some sensible algorithm, it is hard to know what will run at any given moment in time.

As you also might be able to tell from this example, threads make life complicated: it is already hard to tell what will run when! Computers are hard enough to understand without concurrency. Unfortunately, with concurrency, it simply gets worse. Much worse.

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns
prints "main: end"		

Figure 26.3: **Thread Trace (1)**

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Figure 26.4: **Thread Trace (2)**

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Figure 26.5: **Thread Trace (3)**

```
#include <stdio.h>
#include <pthread.h>
#include "mythreads.h"

static volatile int counter = 0;

//
// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

//
// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

Figure 26.6: **Sharing Data: Uh Oh (t1.c)**

26.3 Why It Gets Worse: Shared Data

The simple thread example we showed above was useful in showing how threads are created and how they can run in different orders depending on how the scheduler decides to run them. What it doesn't show you, though, is how threads interact when they access shared data.

Let us imagine a simple example where two threads wish to update a global shared variable. The code we'll study is in Figure 26.6 (page 6).

Here are a few notes about the code. First, as Stevens suggests [SR05], we wrap the thread creation and join routines to simply exit on failure; for a program as simple as this one, we want to at least notice an error occurred (if it did), but not do anything very smart about it (e.g., just exit). Thus, `Pthread_create()` simply calls `pthread_create()` and makes sure the return code is 0; if it isn't, `Pthread_create()` just prints a message and exits.

Second, instead of using two separate function bodies for the worker threads, we just use a single piece of code, and pass the thread an argument (in this case, a string) so we can have each thread print a different letter before its messages.

Finally, and most importantly, we can now look at what each worker is trying to do: add a number to the shared variable `counter`, and do so 10 million times (1e7) in a loop. Thus, the desired final result is: 20,000,000.

We now compile and run the program, to see how it behaves. Sometimes, everything works how we might expect:

```
prompt> gcc -o main main.c -Wall -pthread
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

Unfortunately, when we run this code, even on a single processor, we don't necessarily get the desired result. Sometimes, we get:

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

Let's try it one more time, just to see if we've gone crazy. After all, aren't computers supposed to produce **deterministic** results, as you have been taught?! Perhaps your professors have been lying to you? *(gasp)*

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

Not only is each run wrong, but also yields a *different* result! A big question remains: why does this happen?

TIP: KNOW AND USE YOUR TOOLS

You should always learn new tools that help you write, debug, and understand computer systems. Here, we use a neat tool called a **disassembler.** When you run a disassembler on an executable, it shows you what assembly instructions make up the program. For example, if we wish to understand the low-level code to update a counter (as in our example), we run `objdump` (Linux) to see the assembly code:

```
prompt> objdump -d main
```

Doing so produces a long listing of all the instructions in the program, neatly labeled (particularly if you compiled with the `-g` flag), which includes symbol information in the program. The `objdump` program is just one of many tools you should learn how to use; a debugger like `gdb`, memory profilers like `valgrind` or `purify`, and of course the compiler itself are others that you should spend time to learn more about; the better you are at using your tools, the better systems you'll be able to build.

26.4 The Heart Of The Problem: Uncontrolled Scheduling

To understand why this happens, we must understand the code sequence that the compiler generates for the update to `counter`. In this case, we wish to simply add a number (1) to `counter`. Thus, the code sequence for doing so might look something like this (in x86);

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

This example assumes that the variable `counter` is located at address 0x8049a1c. In this three-instruction sequence, the x86 `mov` instruction is used first to get the memory value at the address and put it into register `eax`. Then, the add is performed, adding 1 (0x1) to the contents of the `eax` register, and finally, the contents of `eax` are stored back into memory at the same address.

Let us imagine one of our two threads (Thread 1) enters this region of code, and is thus about to increment `counter` by one. It loads the value of `counter` (let's say it's 50 to begin with) into its register `eax`. Thus, `eax=50` for Thread 1. Then it adds one to the register; thus `eax=51`. Now, something unfortunate happens: a timer interrupt goes off; thus, the OS saves the state of the currently running thread (its PC, its registers including `eax`, etc.) to the thread's TCB.

Now something worse happens: Thread 2 is chosen to run, and it enters this same piece of code. It also executes the first instruction, getting the value of `counter` and putting it into its `eax` (remember: each thread when running has its own private registers; the registers are **virtualized** by the context-switch code that saves and restores them). The value of

OS	Thread 1	Thread 2	PC	(after instruction) %eax	counter
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	**50**	50
	add $0x1, %eax		108	**51**	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	**50**	50
		add $0x1, %eax	108	**51**	50
		mov %eax, 0x8049a1c	113	51	**51**
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

Figure 26.7: **The Problem: Up Close and Personal**

`counter` is still 50 at this point, and thus Thread 2 has `eax=50`. Let's then assume that Thread 2 executes the next two instructions, incrementing `eax` by 1 (thus `eax=51`), and then saving the contents of `eax` into `counter` (address 0x8049a1c). Thus, the global variable `counter` now has the value 51.

Finally, another context switch occurs, and Thread 1 resumes running. Recall that it had just executed the `mov` and `add`, and is now about to perform the final `mov` instruction. Recall also that `eax=51`. Thus, the final `mov` instruction executes, and saves the value to memory; the counter is set to 51 again.

Put simply, what has happened is this: the code to increment `counter` has been run twice, but `counter`, which started at 50, is now only equal to 51. A "correct" version of this program should have resulted in the variable `counter` equal to 52.

Let's look at a detailed execution trace to understand the problem better. Assume, for this example, that the above code is loaded at address 100 in memory, like the following sequence (note for those of you used to nice, RISC-like instruction sets: x86 has variable-length instructions; this `mov` instruction takes up 5 bytes of memory, and the `add` only 3):

```
100 mov    0x8049a1c, %eax
105 add    $0x1, %eax
108 mov    %eax, 0x8049a1c
```

With these assumptions, what happens is shown in Figure 26.7. Assume the counter starts at value 50, and trace through this example to make sure you understand what is going on.

What we have demonstrated here is called a **race condition**: the results depend on the timing execution of the code. With some bad luck (i.e., context switches that occur at untimely points in the execution), we get the wrong result. In fact, we may get a different result each time; thus, instead of a nice **deterministic** computation (which we are used to from computers), we call this result **indeterminate**, where it is not known what the output will be and it is indeed likely to be different across runs.

Because multiple threads executing this code can result in a race condition, we call this code a **critical section**. A critical section is a piece of code that accesses a shared variable (or more generally, a shared resource) and must not be concurrently executed by more than one thread.

What we really want for this code is what we call **mutual exclusion**. This property guarantees that if one thread is executing within the critical section, the others will be prevented from doing so.

Virtually all of these terms, by the way, were coined by Edsger Dijkstra, who was a pioneer in the field and indeed won the Turing Award because of this and other work; see his 1968 paper on "Cooperating Sequential Processes" [D68] for an amazingly clear description of the problem. We'll be hearing more about Dijkstra in this section of the book.

26.5 The Wish For Atomicity

One way to solve this problem would be to have more powerful instructions that, in a single step, did exactly whatever we needed done and thus removed the possibility of an untimely interrupt. For example, what if we had a super instruction that looked like this?

```
memory-add 0x8049a1c, $0x1
```

Assume this instruction adds a value to a memory location, and the hardware guarantees that it executes **atomically**; when the instruction executed, it would perform the update as desired. It could not be interrupted mid-instruction, because that is precisely the guarantee we receive from the hardware: when an interrupt occurs, either the instruction has not run at all, or it has run to completion; there is no in-between state. Hardware can be a beautiful thing, no?

Atomically, in this context, means "as a unit", which sometimes we take as "all or none." What we'd like is to execute the three instruction sequence atomically:

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

As we said, if we had a single instruction to do this, we could just issue that instruction and be done. But in the general case, we won't have such an instruction. Imagine we were building a concurrent B-tree, and wished to update it; would we really want the hardware to support an "atomic update of B-tree" instruction? Probably not, at least in a sane instruction set.

Thus, what we will instead do is ask the hardware for a few useful instructions upon which we can build a general set of what we call **synchronization primitives**. By using these hardware synchronization primitives, in combination with some help from the operating system, we will be able to build multi-threaded code that accesses critical sections in a

TIP: USE ATOMIC OPERATIONS

Atomic operations are one of the most powerful underlying techniques in building computer systems, from the computer architecture, to concurrent code (what we are studying here), to file systems (which we'll study soon enough), database management systems, and even distributed systems [L+93].

The idea behind making a series of actions **atomic** is simply expressed with the phrase "all or nothing"; it should either appear as if all of the actions you wish to group together occurred, or that none of them occurred, with no in-between state visible. Sometimes, the grouping of many actions into a single atomic action is called a **transaction**, an idea developed in great detail in the world of databases and transaction processing [GR92].

In our theme of exploring concurrency, we'll be using synchronization primitives to turn short sequences of instructions into atomic blocks of execution, but the idea of atomicity is much bigger than that, as we will see. For example, file systems use techniques such as journaling or copy-on-write in order to atomically transition their on-disk state, critical for operating correctly in the face of system failures. If that doesn't make sense, don't worry — it will, in some future chapter.

synchronized and controlled manner, and thus reliably produces the correct result despite the challenging nature of concurrent execution. Pretty awesome, right?

This is the problem we will study in this section of the book. It is a wonderful and hard problem, and should make your mind hurt (a bit). If it doesn't, then you don't understand! Keep working until your head hurts; you then know you're headed in the right direction. At that point, take a break; we don't want your head hurting too much.

THE CRUX:
HOW TO PROVIDE SUPPORT FOR SYNCHRONIZATION

What support do we need from the hardware in order to build useful synchronization primitives? What support do we need from the OS? How can we build these primitives correctly and efficiently? How can programs use them to get the desired results?

26.6 One More Problem: Waiting For Another

This chapter has set up the problem of concurrency as if only one type of interaction occurs between threads, that of accessing shared variables and the need to support atomicity for critical sections. As it turns out, there is another common interaction that arises, where one thread must wait for another to complete some action before it continues. This interaction arises, for example, when a process performs a disk I/O and is put to sleep; when the I/O completes, the process needs to be roused from its slumber so it can continue.

Thus, in the coming chapters, we'll be not only studying how to build support for synchronization primitives to support atomicity but also for mechanisms to support this type of sleeping/waking interaction that is common in multi-threaded programs. If this doesn't make sense right now, that is OK! It will soon enough, when you read the chapter on **condition variables**. If it doesn't by then, well, then it is less OK, and you should read that chapter again (and again) until it does make sense.

26.7 Summary: Why in OS Class?

Before wrapping up, one question that you might have is: why are we studying this in OS class? "History" is the one-word answer; the OS was the first concurrent program, and many techniques were created for use *within* the OS. Later, with multi-threaded processes, application programmers also had to consider such things.

For example, imagine the case where there are two processes running. Assume they both call `write()` to write to the file, and both wish to append the data to the file (i.e., add the data to the end of the file, thus increasing its length). To do so, both must allocate a new block, record in the inode of the file where this block lives, and change the size of the file to reflect the new larger size (among other things; we'll learn more about files in the third part of the book). Because an interrupt may occur at any time, the code that updates these shared structures (e.g., a bitmap for allocation, or the file's inode) are critical sections; thus, OS designers, from the very beginning of the introduction of the interrupt, had to worry about how the OS updates internal structures. An untimely interrupt causes all of the problems described above. Not surprisingly, page tables, process lists, file system structures, and virtually every kernel data structure has to be carefully accessed, with the proper synchronization primitives, to work correctly.

ASIDE: **KEY CONCURRENCY TERMS**
CRITICAL SECTION, RACE CONDITION,
INDETERMINATE, MUTUAL EXCLUSION

These four terms are so central to concurrent code that we thought it worth while to call them out explicitly. See some of Dijkstra's early work [D65,D68] for more details.

- A **critical section** is a piece of code that accesses a *shared* resource, usually a variable or data structure.
- A **race condition** arises if multiple threads of execution enter the critical section at roughly the same time; both attempt to update the shared data structure, leading to a surprising (and perhaps undesirable) outcome.
- An **indeterminate** program consists of one or more race conditions; the output of the program varies from run to run, depending on which threads ran when. The outcome is thus not **deterministic**, something we usually expect from computer systems.
- To avoid these problems, threads should use some kind of **mutual exclusion** primitives; doing so guarantees that only a single thread ever enters a critical section, thus avoiding races, and resulting in deterministic program outputs.

References

[D65] "Solution of a problem in concurrent programming control"
E. W. Dijkstra
Communications of the ACM, 8(9):569, September 1965
Pointed to as the first paper of Dijkstra's where he outlines the mutual exclusion problem and a solution. The solution, however, is not widely used; advanced hardware and OS support is needed, as we will see in the coming chapters.

[D68] "Cooperating sequential processes"
Edsger W. Dijkstra, 1968
Available: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF
Dijkstra has an amazing number of his old papers, notes, and thoughts recorded (for posterity) on this website at the last place he worked, the University of Texas. Much of his foundational work, however, was done years earlier while he was at the Technische Hochshule of Eindhoven (THE), including this famous paper on "cooperating sequential processes", which basically outlines all of the thinking that has to go into writing multi-threaded programs. Dijkstra discovered much of this while working on an operating system named after his school: the "THE" operating system (said "T", "H", "E", and not like the word "the").

[GR92] "Transaction Processing: Concepts and Techniques"
Jim Gray and Andreas Reuter
Morgan Kaufmann, September 1992
This book is the bible of transaction processing, written by one of the legends of the field, Jim Gray. It is, for this reason, also considered Jim Gray's "brain dump", in which he wrote down everything he knows about how database management systems work. Sadly, Gray passed away tragically a few years back, and many of us lost a friend and great mentor, including the co-authors of said book, who were lucky enough to interact with Gray during their graduate school years.

[L+93] "Atomic Transactions"
Nancy Lynch, Michael Merritt, William Weihl, Alan Fekete
Morgan Kaufmann, August 1993
A nice text on some of the theory and practice of atomic transactions for distributed systems. Perhaps a bit formal for some, but lots of good material is found herein.

[SR05] "Advanced Programming in the UNIX Environment"
W. Richard Stevens and Stephen A. Rago
Addison-Wesley, 2005
As we've said many times, buy this book, and read it, in little chunks, preferably before going to bed. This way, you will actually fall asleep more quickly; more importantly, you learn a little more about how to become a serious UNIX programmer.

Homework

This program, `x86.py`, allows you to see how different thread interleavings either cause or avoid race conditions. See the README for details on how the program works and its basic inputs, then answer the questions below.

Questions

1. To start, let's examine a simple program, "loop.s". First, just look at the program, and see if you can understand it: `cat loop.s`. Then, run it with these arguments:

   ```
   ./x86.py -p loop.s -t 1 -i 100 -R dx
   ```

   This specifies a single thread, an interrupt every 100 instructions, and tracing of register `%dx`. Can you figure out what the value of `%dx` will be during the run? Once you have, run the same above and use the `-c` flag to check your answers; note the answers, on the left, show the value of the register (or memory value) *after* the instruction on the right has run.

2. Now run the same code but with these flags:

   ```
   ./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx
   ```

   This specifies two threads, and initializes each %dx register to 3. What values will `%dx` see? Run with the `-c` flag to see the answers. Does the presence of multiple threads affect anything about your calculations? Is there a race condition in this code?

3. Now run the following:

   ```
   ./x86.py -p loop.s -t 2 -i 3 -r -a dx=3,dx=3 -R dx
   ```

   This makes the interrupt interval quite small and random; use different seeds with `-s` to see different interleavings. Does the frequency of interruption change anything about this program?

4. Next we'll examine a different program (`looping-race-nolock.s`). This program accesses a shared variable located at memory address 2000; we'll call this variable `x` for simplicity. Run it with a single thread and make sure you understand what it does, like this:

   ```
   ./x86.py -p looping-race-nolock.s -t 1 -M 2000
   ```

   What value is found in `x` (i.e., at memory address 2000) throughout the run? Use `-c` to check your answer.

5. Now run with multiple iterations and threads:

```
./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000
```

   Do you understand why the code in each thread loops three times? What will the final value of `x` be?

6. Now run with random interrupt intervals:

```
./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0
```

   Then change the random seed, setting `-s 1`, then `-s 2`, etc. Can you tell, just by looking at the thread interleaving, what the final value of `x` will be? Does the exact location of the interrupt matter? Where can it safely occur? Where does an interrupt cause trouble? In other words, where is the critical section exactly?

7. Now use a fixed interrupt interval to explore the program further. Run:

```
./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1
```

   See if you can guess what the final value of the shared variable `x` will be. What about when you change `-i 2`, `-i 3`, etc.? For which interrupt intervals does the program give the "correct" final answer?

8. Now run the same code for more loops (e.g., set `-a bx=100`). What interrupt intervals, set with the `-i` flag, lead to a "correct" outcome? Which intervals lead to surprising results?

9. We'll examine one last program in this homework (`wait-for-me.s`). Run the code like this:

```
./x86.py -p wait-for-me.s -a ax=1,ax=0 -R ax -M 2000
```

   This sets the `%ax` register to 1 for thread 0, and 0 for thread 1, and watches the value of `%ax` and memory location 2000 throughout the run. How should the code behave? How is the value at location 2000 being used by the threads? What will its final value be?

10. Now switch the inputs:

```
./x86.py -p wait-for-me.s -a ax=0,ax=1 -R ax -M 2000
```

   How do the threads behave? What is thread 0 doing? How would changing the interrupt interval (e.g., `-i 1000`, or perhaps to use random intervals) change the trace outcome? Is the program efficiently using the CPU?

27

Interlude: Thread API

This chapter briefly covers the main portions of the thread API. Each part will be explained further in the subsequent chapters, as we show how to use the API. More details can be found in various books and online sources [B89, B97, B+96, K+96]. We should note that the subsequent chapters introduce the concepts of locks and condition variables more slowly, with many examples; this chapter is thus better used as a reference.

CRUX: HOW TO CREATE AND CONTROL THREADS

What interfaces should the OS present for thread creation and control? How should these interfaces be designed to enable ease of use as well as utility?

27.1 Thread Creation

The first thing you have to be able to do to write a multi-threaded program is to create new threads, and thus some kind of thread creation interface must exist. In POSIX, it is easy:

```
#include <pthread.h>
int
pthread_create(      pthread_t *       thread,
               const pthread_attr_t *  attr,
                     void *            (*start_routine)(void*),
                     void *            arg);
```

This declaration might look a little complex (particularly if you haven't used function pointers in C), but actually it's not too bad. There are four arguments: `thread`, `attr`, `start_routine`, and `arg`. The first, `thread`, is a pointer to a structure of type `pthread_t`; we'll use this structure to interact with this thread, and thus we need to pass it to `pthread_create()` in order to initialize it.

The second argument, `attr`, is used to specify any attributes this thread might have. Some examples include setting the stack size or perhaps information about the scheduling priority of the thread. An attribute is initialized with a separate call to `pthread_attr_init()`; see the manual page for details. However, in most cases, the defaults will be fine; in this case, we will simply pass the value `NULL` in.

The third argument is the most complex, but is really just asking: which function should this thread start running in? In C, we call this a **function pointer**, and this one tells us the following is expected: a function name (`start_routine`), which is passed a single argument of type `void *` (as indicated in the parentheses after `start_routine`), and which returns a value of type `void *` (i.e., a **void pointer**).

If this routine instead required an integer argument, instead of a void pointer, the declaration would look like this:

```
int pthread_create(..., // first two args are the same
                   void *    (*start_routine)(int),
                   int       arg);
```

If instead the routine took a void pointer as an argument, but returned an integer, it would look like this:

```
int pthread_create(..., // first two args are the same
                   int       (*start_routine)(void *),
                   void *     arg);
```

Finally, the fourth argument, `arg`, is exactly the argument to be passed to the function where the thread begins execution. You might ask: why do we need these void pointers? Well, the answer is quite simple: having a void pointer as an argument to the function `start_routine` allows us to pass in *any* type of argument; having it as a return value allows the thread to return *any* type of result.

Let's look at an example in Figure 27.1. Here we just create a thread that is passed two arguments, packaged into a single type we define ourselves (`myarg_t`). The thread, once created, can simply cast its argument to the type it expects and thus unpack the arguments as desired.

And there it is! Once you create a thread, you really have another live executing entity, complete with its own call stack, running within the *same* address space as all the currently existing threads in the program. The fun thus begins!

27.2 Thread Completion

The example above shows how to create a thread. However, what happens if you want to wait for a thread to complete? You need to do something special in order to wait for completion; in particular, you must call the routine `pthread_join()`.

```
int pthread_join(pthread_t thread, void **value_ptr);
```

```
1  #include <pthread.h>
2
3  typedef struct __myarg_t {
4      int a;
5      int b;
6  } myarg_t;
7
8  void *mythread(void *arg) {
9      myarg_t *m = (myarg_t *) arg;
10     printf("%d %d\n", m->a, m->b);
11     return NULL;
12 }
13
14 int
15 main(int argc, char *argv[]) {
16     pthread_t p;
17     int rc;
18
19     myarg_t args;
20     args.a = 10;
21     args.b = 20;
22     rc = pthread_create(&p, NULL, mythread, &args);
23     ...
24 }
```

Figure 27.1: **Creating a Thread**

This routine takes two arguments. The first is of type `pthread_t`, and is used to specify which thread to wait for. This variable is initialized by the thread creation routine (when you pass a pointer to it as an argument to `pthread_create()`); if you keep it around, you can use it to wait for that thread to terminate.

The second argument is a pointer to the return value you expect to get back. Because the routine can return anything, it is defined to return a pointer to void; because the `pthread_join()` routine *changes* the value of the passed in argument, you need to pass in a pointer to that value, not just the value itself.

Let's look at another example (Figure 27.2). In the code, a single thread is again created, and passed a couple of arguments via the `myarg_t` structure. To return values, the `myret_t` type is used. Once the thread is finished running, the main thread, which has been waiting inside of the `pthread_join()` routine[1], then returns, and we can access the values returned from the thread, namely whatever is in `myret_t`.

A few things to note about this example. First, often times we don't have to do all of this painful packing and unpacking of arguments. For example, if we just create a thread with no arguments, we can pass `NULL` in as an argument when the thread is created. Similarly, we can pass `NULL` into `pthread_join()` if we don't care about the return value.

Second, if we are just passing in a single value (e.g., an int), we don't

[1]Note we use wrapper functions here; specifically, we call Malloc(), Pthread_join(), and Pthread_create(), which just call their similarly-named lower-case versions and make sure the routines did not return anything unexpected.

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

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

typedef struct __myret_t {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t *r = Malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}

int
main(int argc, char *argv[]) {
    int rc;
    pthread_t p;
    myret_t *m;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    Pthread_create(&p, NULL, mythread, &args);
    Pthread_join(p, (void **) &m);
    printf("returned %d %d\n", m->x, m->y);
    return 0;
}
```

Figure 27.2: **Waiting for Thread Completion**

have to package it up as an argument. Figure 27.3 shows an example. In this case, life is a bit simpler, as we don't have to package arguments and return values inside of structures.

Third, we should note that one has to be extremely careful with how values are returned from a thread. In particular, never return a pointer which refers to something allocated on the thread's call stack. If you do, what do you think will happen? (think about it!) Here is an example of a dangerous piece of code, modified from the example in Figure 27.2.

```
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t r; // ALLOCATED ON STACK: BAD!
    r.x = 1;
    r.y = 2;
    return (void *) &r;
}
```

```
void *mythread(void *arg) {
    int m = (int) arg;
    printf("%d\n", m);
    return (void *) (arg + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc, m;
    Pthread_create(&p, NULL, mythread, (void *) 100);
    Pthread_join(p, (void **) &m);
    printf("returned %d\n", m);
    return 0;
}
```

Figure 27.3: **Simpler Argument Passing to a Thread**

In this case, the variable `r` is allocated on the stack of `mythread`. However, when it returns, the value is automatically deallocated (that's why the stack is so easy to use, after all!), and thus, passing back a pointer to a now deallocated variable will lead to all sorts of bad results. Certainly, when you print out the values you think you returned, you'll probably (but not necessarily!) be surprised. Try it and find out for yourself[2]!

Finally, you might notice that the use of `pthread_create()` to create a thread, followed by an immediate call to `pthread_join()`, is a pretty strange way to create a thread. In fact, there is an easier way to accomplish this exact task; it's called a **procedure call**. Clearly, we'll usually be creating more than just one thread and waiting for it to complete, otherwise there is not much purpose to using threads at all.

We should note that not all code that is multi-threaded uses the join routine. For example, a multi-threaded web server might create a number of worker threads, and then use the main thread to accept requests and pass them to the workers, indefinitely. Such long-lived programs thus may not need to join. However, a parallel program that creates threads to execute a particular task (in parallel) will likely use join to make sure all such work completes before exiting or moving onto the next stage of computation.

27.3 Locks

Beyond thread creation and join, probably the next most useful set of functions provided by the POSIX threads library are those for providing mutual exclusion to a critical section via **locks**. The most basic pair of routines to use for this purpose is provided by this pair of routines:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

[2]Fortunately the compiler `gcc` will likely complain when you write code like this, which is yet another reason to pay attention to compiler warnings.

The routines should be easy to understand and use. When you have a region of code you realize is a **critical section**, and thus needs to be protected by locks in order to operate as desired. You can probably imagine what the code looks like:

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

The intent of the code is as follows: if no other thread holds the lock when `pthread_mutex_lock()` is called, the thread will acquire the lock and enter the critical section. If another thread does indeed hold the lock, the thread trying to grab the lock will not return from the call until it has acquired the lock (implying that the thread holding the lock has released it via the unlock call). Of course, many threads may be stuck waiting inside the lock acquisition function at a given time; only the thread with the lock acquired, however, should call unlock.

Unfortunately, this code is broken, in two important ways. The first problem is a **lack of proper initialization**. All locks must be properly initialized in order to guarantee that they have the correct values to begin with and thus work as desired when lock and unlock are called.

With POSIX threads, there are two ways to initialize locks. One way to do this is to use `PTHREAD_MUTEX_INITIALIZER`, as follows:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Doing so sets the lock to the default values and thus makes the lock usable. The dynamic way to do it (i.e., at run time) is to make a call to `pthread_mutex_init()`, as follows:

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

The first argument to this routine is the address of the lock itself, whereas the second is an optional set of attributes. Read more about the attributes yourself; passing `NULL` in simply uses the defaults. Either way works, but we usually use the dynamic (latter) method. Note that a corresponding call to `pthread_mutex_destroy()` should also be made, when you are done with the lock; see the manual page for all of details.

The second problem with the code above is that it fails to check error codes when calling lock and unlock. Just like virtually any library routine you call in a UNIX system, these routines can also fail! If your code doesn't properly check error codes, the failure will happen silently, which in this case could allow multiple threads into a critical section. Minimally, use wrappers, which assert that the routine succeeded (e.g., as in Figure 27.4); more sophisticated (non-toy) programs, which can't simply exit when something goes wrong, should check for failure and do something appropriate when the lock or unlock does not succeed.

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
  int rc = pthread_mutex_lock(mutex);
  assert(rc == 0);
}
```

Figure 27.4: **An Example Wrapper**

The lock and unlock routines are not the only routines within the pthreads library to interact with locks. In particular, here are two more routines which may be of interest:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            struct timespec *abs_timeout);
```

These two calls are used in lock acquisition. The `trylock` version returns failure if the lock is already held; the `timedlock` version of acquiring a lock returns after a timeout or after acquiring the lock, whichever happens first. Thus, the timedlock with a timeout of zero degenerates to the trylock case. Both of these versions should generally be avoided; however, there are a few cases where avoiding getting stuck (perhaps indefinitely) in a lock acquisition routine can be useful, as we'll see in future chapters (e.g., when we study deadlock).

27.4 Condition Variables

The other major component of any threads library, and certainly the case with POSIX threads, is the presence of a **condition variable**. Condition variables are useful when some kind of signaling must take place between threads, if one thread is waiting for another to do something before it can continue. Two primary routines are used by programs wishing to interact in this way:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

To use a condition variable, one has to in addition have a lock that is associated with this condition. When calling either of the above routines, this lock should be held.

The first routine, `pthread_cond_wait()`, puts the calling thread to sleep, and thus waits for some other thread to signal it, usually when something in the program has changed that the now-sleeping thread might care about. A typical usage looks like this:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;

Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

In this code, after initialization of the relevant lock and condition[3], a thread checks to see if the variable `ready` has yet been set to something other than zero. If not, the thread simply calls the wait routine in order to sleep until some other thread wakes it.

The code to wake a thread, which would run in some other thread, looks like this:

```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

A few things to note about this code sequence. First, when signaling (as well as when modifying the global variable `ready`), we always make sure to have the lock held. This ensures that we don't accidentally introduce a race condition into our code.

Second, you might notice that the wait call takes a lock as its second parameter, whereas the signal call only takes a condition. The reason for this difference is that the wait call, in addition to putting the calling thread to sleep, *releases* the lock when putting said caller to sleep. Imagine if it did not: how could the other thread acquire the lock and signal it to wake up? However, *before* returning after being woken, the `pthread_cond_wait()` re-acquires the lock, thus ensuring that any time the waiting thread is running between the lock acquire at the beginning of the wait sequence, and the lock release at the end, it holds the lock.

One last oddity: the waiting thread re-checks the condition in a while loop, instead of a simple if statement. We'll discuss this issue in detail when we study condition variables in a future chapter, but in general, using a while loop is the simple and safe thing to do. Although it rechecks the condition (perhaps adding a little overhead), there are some pthread implementations that could spuriously wake up a waiting thread; in such a case, without rechecking, the waiting thread will continue thinking that the condition has changed even though it has not. It is safer thus to view waking up as a hint that something might have changed, rather than an absolute fact.

Note that sometimes it is tempting to use a simple flag to signal between two threads, instead of a condition variable and associated lock. For example, we could rewrite the waiting code above to look more like this in the waiting code:

```
while (ready == 0)
    ; // spin
```

The associated signaling code would look like this:

```
ready = 1;
```

[3] Note that one could use `pthread_cond_init()` (and corresponding the `pthread_cond_destroy()` call) instead of the static initializer `PTHREAD_COND_INITIALIZER`. Sound like more work? It is.

Don't ever do this, for the following reasons. First, it performs poorly in many cases (spinning for a long time just wastes CPU cycles). Second, it is error prone. As recent research shows [X+10], it is surprisingly easy to make mistakes when using flags (as above) to synchronize between threads; in that study, roughly half the uses of these *ad hoc* synchronizations were buggy! Don't be lazy; use condition variables even when you think you can get away without doing so.

If condition variables sound confusing, don't worry too much (yet) – we'll be covering them in great detail in a subsequent chapter. Until then, it should suffice to know that they exist and to have some idea how and why they are used.

27.5 Compiling and Running

All of the code examples in this chapter are relatively easy to get up and running. To compile them, you must include the header `pthread.h` in your code. On the link line, you must also explicitly link with the pthreads library, by adding the `-pthread` flag.

For example, to compile a simple multi-threaded program, all you have to do is the following:

```
prompt> gcc -o main main.c -Wall -pthread
```

As long as `main.c` includes the pthreads header, you have now successfully compiled a concurrent program. Whether it works or not, as usual, is a different matter entirely.

27.6 Summary

We have introduced the basics of the pthread library, including thread creation, building mutual exclusion via locks, and signaling and waiting via condition variables. You don't need much else to write robust and efficient multi-threaded code, except patience and a great deal of care!

We now end the chapter with a set of tips that might be useful to you when you write multi-threaded code (see the aside on the following page for details). There are other aspects of the API that are interesting; if you want more information, type `man -k pthread` on a Linux system to see over one hundred APIs that make up the entire interface. However, the basics discussed herein should enable you to build sophisticated (and hopefully, correct and performant) multi-threaded programs. The hard part with threads is not the APIs, but rather the tricky logic of how you build concurrent programs. Read on to learn more.

ASIDE: **THREAD API GUIDELINES**

There are a number of small but important things to remember when you use the POSIX thread library (or really, any thread library) to build a multi-threaded program. They are:

- **Keep it simple.** Above all else, any code to lock or signal between threads should be as simple as possible. Tricky thread interactions lead to bugs.
- **Minimize thread interactions.** Try to keep the number of ways in which threads interact to a minimum. Each interaction should be carefully thought out and constructed with tried and true approaches (many of which we will learn about in the coming chapters).
- **Initialize locks and condition variables.** Failure to do so will lead to code that sometimes works and sometimes fails in very strange ways.
- **Check your return codes.** Of course, in any C and UNIX programming you do, you should be checking each and every return code, and it's true here as well. Failure to do so will lead to bizarre and hard to understand behavior, making you likely to (a) scream, (b) pull some of your hair out, or (c) both.
- **Be careful with how you pass arguments to, and return values from, threads.** In particular, any time you are passing a reference to a variable allocated on the stack, you are probably doing something wrong.
- **Each thread has its own stack.** As related to the point above, please remember that each thread has its own stack. Thus, if you have a locally-allocated variable inside of some function a thread is executing, it is essentially *private* to that thread; no other thread can (easily) access it. To share data between threads, the values must be in the **heap** or otherwise some locale that is globally accessible.
- **Always use condition variables to signal between threads.** While it is often tempting to use a simple flag, don't do it.
- **Use the manual pages.** On Linux, in particular, the pthread man pages are highly informative and discuss much of the nuances presented here, often in even more detail. Read them carefully!

References

[B89] "An Introduction to Programming with Threads"
Andrew D. Birrell
DEC Technical Report,January, 1989
Available: https://birrell.org/andrew/papers/035-Threads.pdf
A classic but older introduction to threaded programming. Still a worthwhile read, and freely available.

[B97] "Programming with POSIX Threads"
David R. Butenhof
Addison-Wesley, May 1997
Another one of these books on threads.

[B+96] "PThreads Programming:
A POSIX Standard for Better Multiprocessing"
Dick Buttlar, Jacqueline Farrell, Bradford Nichols
O'Reilly, September 1996
A reasonable book from the excellent, practical publishing house O'Reilly. Our bookshelves certainly contain a great deal of books from this company, including some excellent offerings on Perl, Python, and Javascript (particularly Crockford's "Javascript: The Good Parts".)

[K+96] "Programming With Threads"
Steve Kleiman, Devang Shah, Bart Smaalders
Prentice Hall, January 1996
Probably one of the better books in this space. Get it at your local library. Or steal it from your mother. More seriously, just ask your mother for it – she'll let you borrow it, don't worry.

[X+10] "Ad Hoc Synchronization Considered Harmful"
Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, Zhiqiang Ma
OSDI 2010, Vancouver, Canada
This paper shows how seemingly simple synchronization code can lead to a surprising number of bugs. Use condition variables and do the signaling correctly!

Homework (Code)

In this section, we'll write some simple multi-threaded programs and use a specific tool, called **helgrind**, to find problems in these programs.

Read the README in the homework download for details on how to build the programs and run `helgrind`.

Questions

1. First build `main-race.c`. Examine the code so you can see the (hopefully obvious) data race in the code. Now run `helgrind` (by typing `valgrind --tool=helgrind main-race`) to see how it reports the race. Does it point to the right lines of code? What other information does it give to you?
2. What happens when you remove one of the offending lines of code? Now add a lock around one of the updates to the shared variable, and then around both. What does `helgrind` report in each of these cases?
3. Now let's look at `main-deadlock.c`. Examine the code. This code has a problem known as **deadlock** (which we discuss in much more depth in a forthcoming chapter). Can you see what problem it might have?
4. Now run `helgrind` on this code. What does `helgrind` report?
5. Now run `helgrind` on `main-deadlock-global.c`. Examine the code; does it have the same problem that `main-deadlock.c` has? Should `helgrind` be reporting the same error? What does this tell you about tools like `helgrind`?
6. Let's next look at `main-signal.c`. This code uses a variable (`done`) to signal that the child is done and that the parent can now continue. Why is this code inefficient? (what does the parent end up spending its time doing, particularly if the child thread takes a long time to complete?)
7. Now run `helgrind` on this program. What does it report? Is the code correct?
8. Now look at a slightly modified version of the code, which is found in `main-signal-cv.c`. This version uses a condition variable to do the signaling (and associated lock). Why is this code preferred to the previous version? Is it correctness, or performance, or both?
9. Once again run `helgrind` on `main-signal-cv`. Does it report any errors?

28

Locks

From the introduction to concurrency, we saw one of the fundamental problems in concurrent programming: we would like to execute a series of instructions atomically, but due to the presence of interrupts on a single processor (or multiple threads executing on multiple processors concurrently), we couldn't. In this chapter, we thus attack this problem directly, with the introduction of something referred to as a **lock**. Programmers annotate source code with locks, putting them around critical sections, and thus ensure that any such critical section executes as if it were a single atomic instruction.

28.1 Locks: The Basic Idea

As an example, assume our critical section looks like this, the canonical update of a shared variable:

```
balance = balance + 1;
```

Of course, other critical sections are possible, such as adding an element to a linked list or other more complex updates to shared structures, but we'll just keep to this simple example for now. To use a lock, we add some code around the critical section like this:

```
1 lock_t mutex; // some globally-allocated lock 'mutex'
2 ...
3 lock(&mutex);
4 balance = balance + 1;
5 unlock(&mutex);
```

A lock is just a variable, and thus to use one, you must declare a **lock variable** of some kind (such as `mutex` above). This lock variable (or just "lock" for short) holds the state of the lock at any instant in time. It is either **available** (or **unlocked** or **free**) and thus no thread holds the lock, or **acquired** (or **locked** or **held**), and thus exactly one thread holds the lock and presumably is in a critical section. We could store other information

in the data type as well, such as which thread holds the lock, or a queue for ordering lock acquisition, but information like that is hidden from the user of the lock.

The semantics of the `lock()` and `unlock()` routines are simple. Calling the routine `lock()` tries to acquire the lock; if no other thread holds the lock (i.e., it is free), the thread will acquire the lock and enter the critical section; this thread is sometimes said to be the **owner** of the lock. If another thread then calls `lock()` on that same lock variable (`mutex` in this example), it will not return while the lock is held by another thread; in this way, other threads are prevented from entering the critical section while the first thread that holds the lock is in there.

Once the owner of the lock calls `unlock()`, the lock is now available (free) again. If no other threads are waiting for the lock (i.e., no other thread has called `lock()` and is stuck therein), the state of the lock is simply changed to free. If there are waiting threads (stuck in `lock()`), one of them will (eventually) notice (or be informed of) this change of the lock's state, acquire the lock, and enter the critical section.

Locks provide some minimal amount of control over scheduling to programmers. In general, we view threads as entities created by the programmer but scheduled by the OS, in any fashion that the OS chooses. Locks yield some of that control back to the programmer; by putting a lock around a section of code, the programmer can guarantee that no more than a single thread can ever be active within that code. Thus locks help transform the chaos that is traditional OS scheduling into a more controlled activity.

28.2 Pthread Locks

The name that the POSIX library uses for a lock is a **mutex**, as it is used to provide **mutual exclusion** between threads, i.e., if one thread is in the critical section, it excludes the others from entering until it has completed the section. Thus, when you see the following POSIX threads code, you should understand that it is doing the same thing as above (we again use our wrappers that check for errors upon lock and unlock):

```
1  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2
3  Pthread_mutex_lock(&lock);   // wrapper for pthread_mutex_lock()
4  balance = balance + 1;
5  Pthread_mutex_unlock(&lock);
```

You might also notice here that the POSIX version passes a variable to lock and unlock, as we may be using *different* locks to protect different variables. Doing so can increase concurrency: instead of one big lock that is used any time any critical section is accessed (a **coarse-grained** locking strategy), one will often protect different data and data structures with different locks, thus allowing more threads to be in locked code at once (a more **fine-grained** approach).

28.3 Building A Lock

By now, you should have some understanding of how a lock works, from the perspective of a programmer. But how should we build a lock? What hardware support is needed? What OS support? It is this set of questions we address in the rest of this chapter.

THE CRUX: HOW TO BUILD A LOCK
How can we build an efficient lock? Efficient locks provided mutual exclusion at low cost, and also might attain a few other properties we discuss below. What hardware support is needed? What OS support?

To build a working lock, we will need some help from our old friend, the hardware, as well as our good pal, the OS. Over the years, a number of different hardware primitives have been added to the instruction sets of various computer architectures; while we won't study how these instructions are implemented (that, after all, is the topic of a computer architecture class), we will study how to use them in order to build a mutual exclusion primitive like a lock. We will also study how the OS gets involved to complete the picture and enable us to build a sophisticated locking library.

28.4 Evaluating Locks

Before building any locks, we should first understand what our goals are, and thus we ask how to evaluate the efficacy of a particular lock implementation. To evaluate whether a lock works (and works well), we should first establish some basic criteria. The first is whether the lock does its basic task, which is to provide **mutual exclusion**. Basically, does the lock work, preventing multiple threads from entering a critical section?

The second is **fairness**. Does each thread contending for the lock get a fair shot at acquiring it once it is free? Another way to look at this is by examining the more extreme case: does any thread contending for the lock **starve** while doing so, thus never obtaining it?

The final criterion is **performance**, specifically the time overheads added by using the lock. There are a few different cases that are worth considering here. One is the case of no contention; when a single thread is running and grabs and releases the lock, what is the overhead of doing so? Another is the case where multiple threads are contending for the lock on a single CPU; in this case, are there performance concerns? Finally, how does the lock perform when there are multiple CPUs involved, and threads on each contending for the lock? By comparing these different scenarios, we can better understand the performance impact of using various locking techniques, as described below.

28.5 Controlling Interrupts

One of the earliest solutions used to provide mutual exclusion was to disable interrupts for critical sections; this solution was invented for single-processor systems. The code would look like this:

```
1 void lock() {
2     DisableInterrupts();
3 }
4 void unlock() {
5     EnableInterrupts();
6 }
```

Assume we are running on such a single-processor system. By turning off interrupts (using some kind of special hardware instruction) before entering a critical section, we ensure that the code inside the critical section will *not* be interrupted, and thus will execute as if it were atomic. When we are finished, we re-enable interrupts (again, via a hardware instruction) and thus the program proceeds as usual.

The main positive of this approach is its simplicity. You certainly don't have to scratch your head too hard to figure out why this works. Without interruption, a thread can be sure that the code it executes will execute and that no other thread will interfere with it.

The negatives, unfortunately, are many. First, this approach requires us to allow any calling thread to perform a *privileged* operation (turning interrupts on and off), and thus *trust* that this facility is not abused. As you already know, any time we are required to trust an arbitrary program, we are probably in trouble. Here, the trouble manifests in numerous ways: a greedy program could call `lock()` at the beginning of its execution and thus monopolize the processor; worse, an errant or malicious program could call `lock()` and go into an endless loop. In this latter case, the OS never regains control of the system, and there is only one recourse: restart the system. Using interrupt disabling as a general-purpose synchronization solution requires too much trust in applications.

Second, the approach does not work on multiprocessors. If multiple threads are running on different CPUs, and each try to enter the same critical section, it does not matter whether interrupts are disabled; threads will be able to run on other processors, and thus could enter the critical section. As multiprocessors are now commonplace, our general solution will have to do better than this.

Third, turning off interrupts for extended periods of time can lead to interrupts becoming lost, which can lead to serious systems problems. Imagine, for example, if the CPU missed the fact that a disk device has finished a read request. How will the OS know to wake the process waiting for said read?

Finally, and probably least important, this approach can be inefficient. Compared to normal instruction execution, code that masks or unmasks interrupts tends to be executed slowly by modern CPUs.

For these reasons, turning off interrupts is only used in limited contexts as a mutual-exclusion primitive. For example, in some cases an

ASIDE: **DEKKER'S AND PETERSON'S ALGORITHMS**
In the 1960's, Dijkstra posed the concurrency problem to his friends, and one of them, a mathematician named Theodorus Jozef Dekker, came up with a solution [D68]. Unlike the solutions we discuss here, which use special hardware instructions and even OS support, **Dekker's algorithm** uses just loads and stores (assuming they are atomic with respect to each other, which was true on early hardware).
Dekker's approach was later refined by Peterson [P81]. Once again, just loads and stores are used, and the idea is to ensure that two threads never enter a critical section at the same time. Here is **Peterson's algorithm** (for two threads); see if you can understand the code. What are the `flag` and `turn` variables used for?

```
int flag[2];
int turn;

void init() {
    flag[0] = flag[1] = 0;    // 1->thread wants to grab lock
    turn = 0;                 // whose turn? (thread 0 or 1?)
}
void lock() {
    flag[self] = 1;           // self: thread ID of caller
    turn = 1 - self;          // make it other thread's turn
    while ((flag[1-self] == 1) && (turn == 1 - self))
        ; // spin-wait
}
void unlock() {
    flag[self] = 0;           // simply undo your intent
}
```

For some reason, developing locks that work without special hardware support became all the rage for a while, giving theory-types a lot of problems to work on. Of course, this line of work became quite useless when people realized it is much easier to assume a little hardware support (and indeed that support had been around from the earliest days of multiprocessing). Further, algorithms like the ones above don't work on modern hardware (due to relaxed memory consistency models), thus making them even less useful than they were before. Yet more research relegated to the dustbin of history...

operating system itself will use interrupt masking to guarantee atomicity when accessing its own data structures, or at least to prevent certain messy interrupt handling situations from arising. This usage makes sense, as the trust issue disappears inside the OS, which always trusts itself to perform privileged operations anyhow.

```
1  typedef struct __lock_t { int flag; } lock_t;
2
3  void init(lock_t *mutex) {
4      // 0 -> lock is available, 1 -> held
5      mutex->flag = 0;
6  }
7
8  void lock(lock_t *mutex) {
9      while (mutex->flag == 1)  // TEST the flag
10         ; // spin-wait (do nothing)
11     mutex->flag = 1;          // now SET it!
12 }
13
14 void unlock(lock_t *mutex) {
15     mutex->flag = 0;
16 }
```

Figure 28.1: **First Attempt: A Simple Flag**

28.6 Test And Set (Atomic Exchange)

Because disabling interrupts does not work on multiple processors, system designers started to invent hardware support for locking. The earliest multiprocessor systems, such as the Burroughs B5000 in the early 1960's [M82], had such support; today all systems provide this type of support, even for single CPU systems.

The simplest bit of hardware support to understand is what is known as a **test-and-set instruction**, also known as **atomic exchange**. To understand how test-and-set works, let's first try to build a simple lock without it. In this failed attempt, we use a simple flag variable to denote whether the lock is held or not.

In this first attempt (Figure 28.1), the idea is quite simple: use a simple variable to indicate whether some thread has possession of a lock. The first thread that enters the critical section will call `lock()`, which **tests** whether the flag is equal to 1 (in this case, it is not), and then **sets** the flag to 1 to indicate that the thread now **holds** the lock. When finished with the critical section, the thread calls `unlock()` and clears the flag, thus indicating that the lock is no longer held.

If another thread happens to call `lock()` while that first thread is in the critical section, it will simply **spin-wait** in the while loop for that thread to call `unlock()` and clear the flag. Once that first thread does so, the waiting thread will fall out of the while loop, set the flag to 1 for itself, and proceed into the critical section.

Unfortunately, the code has two problems: one of correctness, and another of performance. The correctness problem is simple to see once you get used to thinking about concurrent programming. Imagine the code interleaving in Figure 28.2 (page 7); assume `flag=0` to begin.

As you can see from this interleaving, with timely (untimely?) interrupts, we can easily produce a case where *both* threads set the flag to 1 and both threads are thus able to enter the critical section. This behavior is what professionals call "bad" – we have obviously failed to provide the most basic requirement: providing mutual exclusion.

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Figure 28.2: **Trace: No Mutual Exclusion**

The performance problem, which we will address more later on, is the fact that the way a thread waits to acquire a lock that is already held: it endlessly checks the value of flag, a technique known as **spin-waiting**. Spin-waiting wastes time waiting for another thread to release a lock. The waste is exceptionally high on a uniprocessor, where the thread that the waiter is waiting for cannot even run (at least, until a context switch occurs)! Thus, as we move forward and develop more sophisticated solutions, we should also consider ways to avoid this kind of waste.

28.7 Building A Working Spin Lock

While the idea behind the example above is a good one, it is not possible to implement without some support from the hardware. Fortunately, some systems provide an instruction to support the creation of simple locks based on this concept. This more powerful instruction has different names — on SPARC, it is the load/store unsigned byte instruction (`ldstub`), whereas on x86, it is the atomic exchange instruction (`xchg`) — but basically does the same thing across platforms, and is generally referred to as **test-and-set**. We define what the test-and-set instruction does with the following C code snippet:

```
1   int TestAndSet(int *old_ptr, int new) {
2       int old = *old_ptr; // fetch old value at old_ptr
3       *old_ptr = new;     // store 'new' into old_ptr
4       return old;         // return the old value
5   }
```

What the test-and-set instruction does is as follows. It returns the old value pointed to by the `ptr`, and simultaneously updates said value to `new`. The key, of course, is that this sequence of operations is performed **atomically**. The reason it is called "test and set" is that it enables you to "test" the old value (which is what is returned) while simultaneously "setting" the memory location to a new value; as it turns out, this slightly more powerful instruction is enough to build a simple **spin lock**, as we now examine in Figure 28.3. Or better yet: figure it out first yourself!

Let's make sure we understand why this lock works. Imagine first the case where a thread calls `lock()` and no other thread currently holds the lock; thus, `flag` should be 0. When the thread calls `TestAndSet(flag,`

```
1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0 indicates that lock is available, 1 that it is held
7      lock->flag = 0;
8  }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

Figure 28.3: **A Simple Spin Lock Using Test-and-set**

`1)`, the routine will return the old value of `flag`, which is 0; thus, the calling thread, which is *testing* the value of flag, will not get caught spinning in the while loop and will acquire the lock. The thread will also atomically *set* the value to 1, thus indicating that the lock is now held. When the thread is finished with its critical section, it calls `unlock()` to set the flag back to zero.

The second case we can imagine arises when one thread already has the lock held (i.e., `flag` is 1). In this case, this thread will call `lock()` and then call `TestAndSet(flag, 1)` as well. This time, `TestAndSet()` will return the old value at flag, which is 1 (because the lock is held), while simultaneously setting it to 1 again. As long as the lock is held by another thread, `TestAndSet()` will repeatedly return 1, and thus this thread will spin and spin until the lock is finally released. When the flag is finally set to 0 by some other thread, this thread will call `TestAndSet()` again, which will now return 0 while atomically setting the value to 1 and thus acquire the lock and enter the critical section.

By making both the **test** (of the old lock value) and **set** (of the new value) a single atomic operation, we ensure that only one thread acquires the lock. And that's how to build a working mutual exclusion primitive!

You may also now understand why this type of lock is usually referred

TIP: THINK ABOUT CONCURRENCY AS MALICIOUS SCHEDULER

From this example, you might get a sense of the approach you need to take to understand concurrent execution. What you should try to do is to pretend you are a **malicious scheduler**, one that interrupts threads at the most inopportune of times in order to foil their feeble attempts at building synchronization primitives. What a mean scheduler you are! Although the exact sequence of interrupts may be *improbable*, it is *possible*, and that is all we need to demonstrate that a particular approach does not work. It can be useful to think maliciously! (at least, sometimes)

to as a **spin lock**. It is the simplest type of lock to build, and simply spins, using CPU cycles, until the lock becomes available. To work correctly on a single processor, it requires a **preemptive scheduler** (i.e., one that will interrupt a thread via a timer, in order to run a different thread, from time to time). Without preemption, spin locks don't make much sense on a single CPU, as a thread spinning on a CPU will never relinquish it.

28.8 Evaluating Spin Locks

Given our basic spin lock, we can now evaluate how effective it is along our previously described axes. The most important aspect of a lock is **correctness**: does it provide mutual exclusion? The answer here is yes: the spin lock only allows a single thread to enter the critical section at a time. Thus, we have a correct lock.

The next axis is **fairness**. How fair is a spin lock to a waiting thread? Can you guarantee that a waiting thread will ever enter the critical section? The answer here, unfortunately, is bad news: spin locks don't provide any fairness guarantees. Indeed, a thread spinning may spin forever, under contention. Spin locks are not fair and may lead to starvation.

The final axis is **performance**. What are the costs of using a spin lock? To analyze this more carefully, we suggest thinking about a few different cases. In the first, imagine threads competing for the lock on a single processor; in the second, consider the threads as spread out across many processors.

For spin locks, in the single CPU case, performance overheads can be quite painful; imagine the case where the thread holding the lock is pre-empted within a critical section. The scheduler might then run every other thread (imagine there are $N - 1$ others), each of which tries to acquire the lock. In this case, each of those threads will spin for the duration of a time slice before giving up the CPU, a waste of CPU cycles.

However, on multiple CPUs, spin locks work reasonably well (if the number of threads roughly equals the number of CPUs). The thinking goes as follows: imagine Thread A on CPU 1 and Thread B on CPU 2, both contending for a lock. If Thread A (CPU 1) grabs the lock, and then Thread B tries to, B will spin (on CPU 2). However, presumably the critical section is short, and thus soon the lock becomes available, and is acquired by Thread B. Spinning to wait for a lock held on another processor doesn't waste many cycles in this case, and thus can be effective.

28.9 Compare-And-Swap

Another hardware primitive that some systems provide is known as the **compare-and-swap** instruction (as it is called on SPARC, for example), or **compare-and-exchange** (as it called on x86). The C pseudocode for this single instruction is found in Figure 28.4.

The basic idea is for compare-and-swap to test whether the value at the

```
1  int CompareAndSwap(int *ptr, int expected, int new) {
2      int actual = *ptr;
3      if (actual == expected)
4          *ptr = new;
5      return actual;
6  }
```

Figure 28.4: **Compare-and-swap**

address specified by `ptr` is equal to `expected`; if so, update the memory location pointed to by `ptr` with the new value. If not, do nothing. In either case, return the actual value at that memory location, thus allowing the code calling compare-and-swap to know whether it succeeded or not.

With the compare-and-swap instruction, we can build a lock in a manner quite similar to that with test-and-set. For example, we could just replace the `lock()` routine above with the following:

```
1  void lock(lock_t *lock) {
2      while (CompareAndSwap(&lock->flag, 0, 1) == 1)
3          ; // spin
4  }
```

The rest of the code is the same as the test-and-set example above. This code works quite similarly; it simply checks if the flag is 0 and if so, atomically swaps in a 1 thus acquiring the lock. Threads that try to acquire the lock while it is held will get stuck spinning until the lock is finally released.

If you want to see how to really make a C-callable x86-version of compare-and-swap, this code sequence might be useful (from [S05]):

```
1  char CompareAndSwap(int *ptr, int old, int new) {
2      unsigned char ret;
3
4      // Note that sete sets a 'byte' not the word
5      __asm__ __volatile__ (
6            "  lock\n"
7            "  cmpxchgl %2,%1\n"
8            "  sete %0\n"
9            : "=q" (ret), "=m" (*ptr)
10           : "r" (new), "m" (*ptr), "a" (old)
11           : "memory");
12     return ret;
13 }
```

Finally, as you may have sensed, compare-and-swap is a more powerful instruction than test-and-set. We will make some use of this power in the future when we briefly delve into topics such as **lock-free synchronization** [H91]. However, if we just build a simple spin lock with it, its behavior is identical to the spin lock we analyzed above.

```
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no one has updated *ptr since the LoadLinked to this address) {
        *ptr = value;
        return 1; // success!
    } else {
        return 0; // failed to update
    }
}
```

Figure 28.5: **Load-linked And Store-conditional**

28.10 Load-Linked and Store-Conditional

Some platforms provide a pair of instructions that work in concert to help build critical sections. On the MIPS architecture [H93], for example, the **load-linked** and **store-conditional** instructions can be used in tandem to build locks and other concurrent structures. The C pseudocode for these instructions is as found in Figure 28.5. Alpha, PowerPC, and ARM provide similar instructions [W09].

The load-linked operates much like a typical load instruction, and simply fetches a value from memory and places it in a register. The key difference comes with the store-conditional, which only succeeds (and updates the value stored at the address just load-linked from) if no intervening store to the address has taken place. In the case of success, the store-conditional returns 1 and updates the value at `ptr` to `value`; if it fails, the value at `ptr` is *not* updated and 0 is returned.

As a challenge to yourself, try thinking about how to build a lock using load-linked and store-conditional. Then, when you are finished, look at the code below which provides one simple solution. Do it! The solution is in Figure 28.6.

The `lock()` code is the only interesting piece. First, a thread spins waiting for the flag to be set to 0 (and thus indicate the lock is not held). Once so, the thread tries to acquire the lock via the store-conditional; if it succeeds, the thread has atomically changed the flag's value to 1 and thus can proceed into the critical section.

```
void lock(lock_t *lock) {
    while (1) {
        while (LoadLinked(&lock->flag) == 1)
            ; // spin until it's zero
        if (StoreConditional(&lock->flag, 1) == 1)
            return; // if set-it-to-1 was a success: all done
                    // otherwise: try it all over again
    }
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Figure 28.6: **Using LL/SC To Build A Lock**

TIP: LESS CODE IS BETTER CODE (LAUER'S LAW)
Programmers tend to brag about how much code they wrote to do something. Doing so is fundamentally broken. What one should brag about, rather, is how *little* code one wrote to accomplish a given task. Short, concise code is always preferred; it is likely easier to understand and has fewer bugs. As Hugh Lauer said, when discussing the construction of the Pilot operating system: "If the same people had twice as much time, they could produce as good of a system in half the code." [L81] We'll call this **Lauer's Law**, and it is well worth remembering. So next time you're bragging about how much code you wrote to finish the assignment, think again, or better yet, go back, rewrite, and make the code as clear and concise as possible.

Note how failure of the store-conditional might arise. One thread calls `lock()` and executes the load-linked, returning 0 as the lock is not held. Before it can attempt the store-conditional, it is interrupted and another thread enters the lock code, also executing the load-linked instruction, and also getting a 0 and continuing. At this point, two threads have each executed the load-linked and each are about to attempt the store-conditional. The key feature of these instructions is that only one of these threads will succeed in updating the flag to 1 and thus acquire the lock; the second thread to attempt the store-conditional will fail (because the other thread updated the value of flag between its load-linked and store-conditional) and thus have to try to acquire the lock again.

In class a few years ago, undergraduate student David Capel suggested a more concise form of the above, for those of you who enjoy short-circuiting boolean conditionals. See if you can figure out why it is equivalent. It certainly is shorter!

```
1  void lock(lock_t *lock) {
2    while (LoadLinked(&lock->flag)||!StoreConditional(&lock->flag, 1))
3      ; // spin
4  }
```

28.11 Fetch-And-Add

One final hardware primitive is the **fetch-and-add** instruction, which atomically increments a value while returning the old value at a particular address. The C pseudocode for the fetch-and-add instruction looks like this:

```
1  int FetchAndAdd(int *ptr) {
2      int old = *ptr;
3      *ptr = old + 1;
4      return old;
5  }
```

```
1  typedef struct __lock_t {
2      int ticket;
3      int turn;
4  } lock_t;
5
6  void lock_init(lock_t *lock) {
7      lock->ticket = 0;
8      lock->turn   = 0;
9  }
10
11 void lock(lock_t *lock) {
12     int myturn = FetchAndAdd(&lock->ticket);
13     while (lock->turn != myturn)
14         ; // spin
15  }
16
17 void unlock(lock_t *lock) {
18     lock->turn = lock->turn + 1;
19 }
```

Figure 28.7: **Ticket Locks**

In this example, we'll use fetch-and-add to build a more interesting **ticket lock**, as introduced by Mellor-Crummey and Scott [MS91]. The lock and unlock code looks like what you see in Figure 28.7.

Instead of a single value, this solution uses a ticket and turn variable in combination to build a lock. The basic operation is pretty simple: when a thread wishes to acquire a lock, it first does an atomic fetch-and-add on the ticket value; that value is now considered this thread's "turn" (`myturn`). The globally shared `lock->turn` is then used to determine which thread's turn it is; when `(myturn == turn)` for a given thread, it is that thread's turn to enter the critical section. Unlock is accomplished simply by incrementing the turn such that the next waiting thread (if there is one) can now enter the critical section.

Note one important difference with this solution versus our previous attempts: it ensures progress for all threads. Once a thread is assigned its ticket value, it will be scheduled at some point in the future (once those in front of it have passed through the critical section and released the lock). In our previous attempts, no such guarantee existed; a thread spinning on test-and-set (for example) could spin forever even as other threads acquire and release the lock.

28.12 Too Much Spinning: What Now?

Our simple hardware-based locks are simple (only a few lines of code) and they work (you could even prove that if you'd like to, by writing some code), which are two excellent properties of any system or code. However, in some cases, these solutions can be quite inefficient. Imagine you are running two threads on a single processor. Now imagine that one thread (thread 0) is in a critical section and thus has a lock held, and unfortunately gets interrupted. The second thread (thread 1) now tries to acquire the lock, but finds that it is held. Thus, it begins to spin. And spin.

Then it spins some more. And finally, a timer interrupt goes off, thread 0 is run again, which releases the lock, and finally (the next time it runs, say), thread 1 won't have to spin so much and will be able to acquire the lock. Thus, any time a thread gets caught spinning in a situation like this, it wastes an entire time slice doing nothing but checking a value that isn't going to change! The problem gets worse with N threads contending for a lock; $N - 1$ time slices may be wasted in a similar manner, simply spinning and waiting for a single thread to release the lock. And thus, our next problem:

THE CRUX: HOW TO AVOID SPINNING
How can we develop a lock that doesn't needlessly waste time spinning on the CPU?

Hardware support alone cannot solve the problem. We'll need OS support too! Let's now figure out just how that might work.

28.13 A Simple Approach: Just Yield, Baby

Hardware support got us pretty far: working locks, and even (as with the case of the ticket lock) fairness in lock acquisition. However, we still have a problem: what to do when a context switch occurs in a critical section, and threads start to spin endlessly, waiting for the interrupted (lock-holding) thread to be run again?

Our first try is a simple and friendly approach: when you are going to spin, instead give up the CPU to another thread. Or, as Al Davis might say, "just yield, baby!" [D91]. Figure 28.8 presents the approach.

In this approach, we assume an operating system primitive `yield()` which a thread can call when it wants to give up the CPU and let another thread run. A thread can be in one of three states (running, ready, or blocked); yield is simply a system call that moves the caller from the

```
1  void init() {
2      flag = 0;
3  }
4
5  void lock() {
6      while (TestAndSet(&flag, 1) == 1)
7          yield(); // give up the CPU
8  }
9
10 void unlock() {
11     flag = 0;
12 }
```

Figure 28.8: **Lock With Test-and-set And Yield**

running state to the **ready** state, and thus promotes another thread to running. Thus, the yielding process essentially **deschedules** itself.

Think about the example with two threads on one CPU; in this case, our yield-based approach works quite well. If a thread happens to call `lock()` and find a lock held, it will simply yield the CPU, and thus the other thread will run and finish its critical section. In this simple case, the yielding approach works well.

Let us now consider the case where there are many threads (say 100) contending for a lock repeatedly. In this case, if one thread acquires the lock and is preempted before releasing it, the other 99 will each call `lock()`, find the lock held, and yield the CPU. Assuming some kind of round-robin scheduler, each of the 99 will execute this run-and-yield pattern before the thread holding the lock gets to run again. While better than our spinning approach (which would waste 99 time slices spinning), this approach is still costly; the cost of a context switch can be substantial, and there is thus plenty of waste.

Worse, we have not tackled the starvation problem at all. A thread may get caught in an endless yield loop while other threads repeatedly enter and exit the critical section. We clearly will need an approach that addresses this problem directly.

28.14 Using Queues: Sleeping Instead Of Spinning

The real problem with our previous approaches is that they leave too much to chance. The scheduler determines which thread runs next; if the scheduler makes a bad choice, a thread runs that must either spin waiting for the lock (our first approach), or yield the CPU immediately (our second approach). Either way, there is potential for waste and no prevention of starvation.

Thus, we must explicitly exert some control over which thread next gets to acquire the lock after the current holder releases it. To do this, we will need a little more OS support, as well as a queue to keep track of which threads are waiting to acquire the lock.

For simplicity, we will use the support provided by Solaris, in terms of two calls: `park()` to put a calling thread to sleep, and `unpark(threadID)` to wake a particular thread as designated by `threadID`. These two routines can be used in tandem to build a lock that puts a caller to sleep if it tries to acquire a held lock and wakes it when the lock is free. Let's look at the code in Figure 28.9 to understand one possible use of such primitives.

We do a couple of interesting things in this example. First, we combine the old test-and-set idea with an explicit queue of lock waiters to make a more efficient lock. Second, we use a queue to help control who gets the lock next and thus avoid starvation.

You might notice how the guard is used (Figure 28.9, page 16), basically as a spin-lock around the flag and queue manipulations the lock is using. This approach thus doesn't avoid spin-waiting entirely; a thread

```
1  typedef struct __lock_t {
2      int flag;
3      int guard;
4      queue_t *q;
5  } lock_t;
6
7  void lock_init(lock_t *m) {
8      m->flag  = 0;
9      m->guard = 0;
10     queue_init(m->q);
11 }
12
13 void lock(lock_t *m) {
14     while (TestAndSet(&m->guard, 1) == 1)
15         ; //acquire guard lock by spinning
16     if (m->flag == 0) {
17         m->flag = 1; // lock is acquired
18         m->guard = 0;
19     } else {
20         queue_add(m->q, gettid());
21         m->guard = 0;
22         park();
23     }
24 }
25
26 void unlock(lock_t *m) {
27     while (TestAndSet(&m->guard, 1) == 1)
28         ; //acquire guard lock by spinning
29     if (queue_empty(m->q))
30         m->flag = 0; // let go of lock; no one wants it
31     else
32         unpark(queue_remove(m->q)); // hold lock (for next thread!)
33     m->guard = 0;
34 }
```

Figure 28.9: **Lock With Queues, Test-and-set, Yield, And Wakeup**

might be interrupted while acquiring or releasing the lock, and thus cause other threads to spin-wait for this one to run again. However, the time spent spinning is quite limited (just a few instructions inside the lock and unlock code, instead of the user-defined critical section), and thus this approach may be reasonable.

Second, you might notice that in `lock()`, when a thread can not acquire the lock (it is already held), we are careful to add ourselves to a queue (by calling the `gettid()` call to get the thread ID of the current thread), set guard to 0, and yield the CPU. A question for the reader: What would happen if the release of the guard lock came *after* the `park()`, and not before? Hint: something bad.

You might also notice the interesting fact that the flag does not get set back to 0 when another thread gets woken up. Why is this? Well, it is not an error, but rather a necessity! When a thread is woken up, it will be as if it is returning from `park()`; however, it does not hold the guard at that point in the code and thus cannot even try to set the flag to 1. Thus, we just pass the lock directly from the thread releasing the lock to the next thread acquiring it; flag is not set to 0 in-between.

ASIDE: **MORE REASON TO AVOID SPINNING: PRIORITY INVERSION**

One good reason to avoid spin locks is performance: as described in the main text, if a thread is interrupted while holding a lock, other threads that use spin locks will spend a large amount of CPU time just waiting for the lock to become available. However, it turns out there is another interesting reason to avoid spin locks on some systems: correctness. The problem to be wary of is known as **priority inversion**, which unfortunately is an intergalactic scourge, occurring on Earth [M15] and Mars [R97]!

Let's assume there are two threads in a system. Thread 2 (T2) has a high scheduling priority, and Thread 1 (T1) has lower priority. In this example, let's assume that the CPU scheduler will always run T2 over T1, if indeed both are runnable; T1 only runs when T2 is not able to do so (e.g., when T2 is blocked on I/O).

Now, the problem. Assume T2 is blocked for some reason. So T1 runs, grabs a spin lock, and enters a critical section. T2 now becomes unblocked (perhaps because an I/O completed), and the CPU scheduler immediately schedules it (thus descheduling T1). T2 now tries to acquire the lock, and because it can't (T1 holds the lock), it just keeps spinning. Because the lock is a spin lock, T2 spins forever, and the system is hung.

Just avoiding the use of spin locks, unfortunately, does not avoid the problem of inversion (alas). Imagine three threads, T1, T2, and T3, with T3 at the highest priority, and T1 the lowest. Imagine now that T1 grabs a lock. T3 then starts, and because it is higher priority than T1, runs immediately (preempting T1). T3 tries to acquire the lock that T1 holds, but gets stuck waiting, because T1 still holds it. If T2 starts to run, it will have higher priority than T1, and thus it will run. T3, which is higher priority than T2, is stuck waiting for T1, which may never run now that T2 is running. Isn't it sad that the mighty T3 can't run, while lowly T2 controls the CPU? Having high priority just ain't what it used to be.

You can address the priority inversion problem in a number of ways. In the specific case where spin locks cause the problem, you can avoid using spin locks (described more below). More generally, a higher-priority thread waiting for a lower-priority thread can temporarily boost the lower thread's priority, thus enabling it to run and overcoming the inversion, a technique known as **priority inheritance**. A last solution is simplest: ensure all threads have the same priority.

Finally, you might notice the perceived race condition in the solution, just before the call to `park()`. With just the wrong timing, a thread will be about to park, assuming that it should sleep until the lock is no longer held. A switch at that time to another thread (say, a thread holding the lock) could lead to trouble, for example, if that thread then released the lock. The subsequent park by the first thread would then sleep forever (potentially), a problem sometimes called the **wakeup/waiting race**.

Solaris solves this problem by adding a third system call: `setpark()`. By calling this routine, a thread can indicate it is *about to* park. If it then happens to be interrupted and another thread calls unpark before park is actually called, the subsequent park returns immediately instead of sleeping. The code modification, inside of `lock()`, is quite small:

```
1          queue_add(m->q, gettid());
2          setpark(); // new code
3          m->guard = 0;
```

A different solution could pass the guard into the kernel. In that case, the kernel could take precautions to atomically release the lock and dequeue the running thread.

28.15 Different OS, Different Support

We have thus far seen one type of support that an OS can provide in order to build a more efficient lock in a thread library. Other OS's provide similar support; the details vary.

For example, Linux provides a **futex** which is similar to the Solaris interface but provides more in-kernel functionality. Specifically, each futex has associated with it a specific physical memory location, as well as a per-futex in-kernel queue. Callers can use futex calls (described below) to sleep and wake as need be.

Specifically, two calls are available. The call to `futex_wait(address, expected)` puts the calling thread to sleep, assuming the value at `address` is equal to `expected`. If it is *not* equal, the call returns immediately. The call to the routine `futex_wake(address)` wakes one thread that is waiting on the queue. The usage of these calls in a Linux mutex is shown in Figure 28.10 (page 19).

This code snippet from `lowlevellock.h` in the nptl library (part of the gnu libc library) [L09] is interesting for a few reasons. First, it uses a single integer to track both whether the lock is held or not (the high bit of the integer) and the number of waiters on the lock (all the other bits). Thus, if the lock is negative, it is held (because the high bit is set and that bit determines the sign of the integer).

Second, the code snippet shows how to optimize for the common case, specifically when there is no contention for the lock; with only one thread acquiring and releasing a lock, very little work is done (the atomic bit test-and-set to lock and an atomic add to release the lock).

See if you can puzzle through the rest of this "real-world" lock to understand how it works. Do it and become a master of Linux locking, or at least somebody who listens when a book tells you to do something [1].

[1] Like buy a print copy of OSTEP! Even though the book is available for free online, wouldn't you just love a hard cover for your desk? Or, better yet, ten copies to share with friends and family? And maybe one extra copy to throw at an enemy? (the book *is* heavy, and thus chucking it is surprisingly effective)

```
1  void mutex_lock (int *mutex) {
2    int v;
3    /* Bit 31 was clear, we got the mutex (this is the fastpath)  */
4    if (atomic_bit_test_set (mutex, 31) == 0)
5      return;
6    atomic_increment (mutex);
7    while (1) {
8        if (atomic_bit_test_set (mutex, 31) == 0) {
9            atomic_decrement (mutex);
10           return;
11       }
12       /* We have to wait now. First make sure the futex value
13          we are monitoring is truly negative (i.e. locked). */
14       v = *mutex;
15       if (v >= 0)
16         continue;
17       futex_wait (mutex, v);
18   }
19 }
20
21 void mutex_unlock (int *mutex) {
22   /* Adding 0x80000000 to the counter results in 0 if and only if
23     there are not other interested threads */
24   if (atomic_add_zero (mutex, 0x80000000))
25     return;
26
27   /* There are other threads waiting for this mutex,
28      wake one of them up.  */
29   futex_wake (mutex);
30 }
```

Figure 28.10: **Linux-based Futex Locks**

28.16 Two-Phase Locks

One final note: the Linux approach has the flavor of an old approach that has been used on and off for years, going at least as far back to Dahm Locks in the early 1960's [M82], and is now referred to as a **two-phase lock**. A two-phase lock realizes that spinning can be useful, particularly if the lock is about to be released. So in the first phase, the lock spins for a while, hoping that it can acquire the lock.

However, if the lock is not acquired during the first spin phase, a second phase is entered, where the caller is put to sleep, and only woken up when the lock becomes free later. The Linux lock above is a form of such a lock, but it only spins once; a generalization of this could spin in a loop for a fixed amount of time before using **futex** support to sleep.

Two-phase locks are yet another instance of a **hybrid** approach, where combining two good ideas may indeed yield a better one. Of course, whether it does depends strongly on many things, including the hardware environment, number of threads, and other workload details. As always, making a single general-purpose lock, good for all possible use cases, is quite a challenge.

28.17 Summary

The above approach shows how real locks are built these days: some hardware support (in the form of a more powerful instruction) plus some operating system support (e.g., in the form of `park()` and `unpark()` primitives on Solaris, or **futex** on Linux). Of course, the details differ, and the exact code to perform such locking is usually highly tuned. Check out the Solaris or Linux code bases if you want to see more details; they are a fascinating read [L09, S09]. Also see David et al.'s excellent work for a comparison of locking strategies on modern multiprocessors [D+13].

References

[D91] "Just Win, Baby: Al Davis and His Raiders"
Glenn Dickey, Harcourt 1991
There is even an undoubtedly bad book about Al Davis and his famous "just win" quote. Or, we suppose, the book is more about Al Davis and the Raiders, and maybe not just the quote. Read the book to find out?

[D+13] "Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask"
Tudor David, Rachid Guerraoui, Vasileios Trigonakis
SOSP '13, Nemacolin Woodlands Resort, Pennsylvania, November 2013
An excellent recent paper comparing many different ways to build locks using hardware primitives. A great read to see how many ideas over the years work on modern hardware.

[D68] "Cooperating sequential processes"
Edsger W. Dijkstra, 1968
Available: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF
One of the early seminal papers in the area. Discusses how Dijkstra posed the original concurrency problem, and Dekker's solution.

[H93] "MIPS R4000 Microprocessor User's Manual"
Joe Heinrich, Prentice-Hall, June 1993
Available: http://cag.csail.mit.edu/raw/documents/R4400_Uman_book_Ed2.pdf

[H91] "Wait-free Synchronization"
Maurice Herlihy
ACM Transactions on Programming Languages and Systems (TOPLAS)
Volume 13, Issue 1, January 1991
A landmark paper introducing a different approach to building concurrent data structures. However, because of the complexity involved, many of these ideas have been slow to gain acceptance in deployed systems.

[L81] "Observations on the Development of an Operating System"
Hugh Lauer
SOSP '81, Pacific Grove, California, December 1981
A must-read retrospective about the development of the Pilot OS, an early PC operating system. Fun and full of insights.

[L09] "glibc 2.9 (include Linux pthreads implementation)"
Available: http://ftp.gnu.org/gnu/glibc/
In particular, take a look at the nptl subdirectory where you will find most of the pthread support in Linux today.

[M82] "The Architecture of the Burroughs B5000
20 Years Later and Still Ahead of the Times?"
Alastair J.W. Mayer, 1982
www.ajwm.net/amayer/papers/B5000.html
From the paper: "One particularly useful instruction is the RDLK (read-lock). It is an indivisible operation which reads from and writes into a memory location." RDLK is thus an early test-and-set primitive, if not the earliest. Some credit here goes to an engineer named Dave Dahm, who apparently invented a number of these things for the Burroughs systems, including a form of spin locks (called "Buzz Locks") as well as a two-phase lock eponymously called "Dahm Locks."

[M15] "OSSpinLock Is Unsafe"
John McCall
Available: mjtsai.com/blog/2015/12/16/osspinlock-is-unsafe
A short post about why calling OSSpinLock on Mac OS X is unsafe when using threads of different priorities – you might end up spinning forever! So be careful, Mac fanatics, even your mighty system sometimes is less than perfect...

[MS91] "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors"
John M. Mellor-Crummey and M. L. Scott
ACM TOCS, Volume 9, Issue 1, February 1991
An excellent and thorough survey on different locking algorithms. However, no operating systems support is used, just fancy hardware instructions.

[P81] "Myths About the Mutual Exclusion Problem"
G.L. Peterson
Information Processing Letters, 12(3), pages 115–116, 1981
Peterson's algorithm introduced here.

[R97] "What Really Happened on Mars?"
Glenn E. Reeves
Available: research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html
A detailed description of priority inversion on the Mars Pathfinder robot. This low-level concurrent code matters a lot, especially in space!

[S05] "Guide to porting from Solaris to Linux on x86"
Ajay Sood, April 29, 2005
Available: `http://www.ibm.com/developerworks/linux/library/l-solar/`

[S09] "OpenSolaris Thread Library"
Available: http://src.opensolaris.org/source/xref/onnv/onnv-gate/
usr/src/lib/libc/port/threads/synch.c
This is also pretty interesting to look at, though who knows what will happen to it now that Oracle owns Sun. Thanks to Mike Swift for the pointer to the code.

[W09] "Load-Link, Store-Conditional"
Wikipedia entry on said topic, as of October 22, 2009
`http://en.wikipedia.org/wiki/Load-Link/Store-Conditional`
Can you believe we referenced wikipedia? Pretty lazy, no? But, we found the information there first, and it felt wrong not to cite it. Further, they even listed the instructions for the different architectures: `ldl_l/stl_c` and `ldq_l/stq_c` (Alpha), `lwarx/stwcx` (PowerPC), `ll/sc` (MIPS), and `ldrex/strex` (ARM version 6 and above). Actually wikipedia is pretty amazing, so don't be so harsh, OK?

[WG00] "The SPARC Architecture Manual: Version 9"
David L. Weaver and Tom Germond, September 2000
SPARC International, San Jose, California
Available: `http://www.sparc.org/standards/SPARCV9.pdf`
Also see: `http://developers.sun.com/solaris/articles/atomic_sparc/` for some more details on Sparc atomic operations.

Homework

This program, `x86.py`, allows you to see how different thread interleavings either cause or avoid race conditions. See the README for details on how the program works and its basic inputs, then answer the questions below.

Questions

1. First let's get ready to run `x86.py` with the flag `-p flag.s`. This code "implements" locking with a single memory flag. Can you understand what the assembly code is trying to do?
2. When you run with the defaults, does `flag.s` work as expected? Does it produce the correct result? Use the `-M` and `-R` flags to trace variables and registers (and turn on `-c` to see their values). Can you predict what value will end up in `flag` as the code runs?
3. Change the value of the register `%bx` with the `-a` flag (e.g., `-a bx=2,bx=2` if you are running just two threads). What does the code do? How does it change your answer for the question above?
4. Set `bx` to a high value for each thread, and then use the `-i` flag to generate different interrupt frequencies; what values lead to a bad outcomes? Which lead to good outcomes?
5. Now let's look at the program `test-and-set.s`. First, try to understand the code, which uses the `xchg` instruction to build a simple locking primitive. How is the lock acquire written? How about lock release?
6. Now run the code, changing the value of the interrupt interval (`-i`) again, and making sure to loop for a number of times. Does the code always work as expected? Does it sometimes lead to an inefficient use of the CPU? How could you quantify that?
7. Use the `-P` flag to generate specific tests of the locking code. For example, run a schedule that grabs the lock in the first thread, but then tries to acquire it in the second. Does the right thing happen? What else should you test?
8. Now let's look at the code in `peterson.s`, which implements Peterson's algorithm (mentioned in a sidebar in the text). Study the code and see if you can make sense of it.
9. Now run the code with different values of `-i`. What kinds of different behavior do you see?
10. Can you control the scheduling (with the `-P` flag) to "prove" that the code works? What are the different cases you should show hold? Think about mutual exclusion and deadlock avoidance.

11. Now study the code for the ticket lock in `ticket.s`. Does it match the code in the chapter?

12. Now run the code, with the following flags: `-a bx=1000,bx=1000` (this flag sets each thread to loop through the critical 1000 times). Watch what happens over time; do the threads spend much time spinning waiting for the lock?

13. How does the code behave as you add more threads?

14. Now examine `yield.s`, in which we pretend that a `yield` instruction enables one thread to yield control of the CPU to another (realistically, this would be an OS primitive, but for the simplicity of simulation, we assume there is an instruction that does the task). Find a scenario where `test-and-set.s` wastes cycles spinning, but `yield.s` does not. How many instructions are saved? In what scenarios do these savings arise?

15. Finally, examine `test-and-test-and-set.s`. What does this lock do? What kind of savings does it introduce as compared to `test-and-set.s`?

29

Lock-based Concurrent Data Structures

Before moving beyond locks, we'll first describe how to use locks in some common data structures. Adding locks to a data structure to make it usable by threads makes the structure **thread safe**. Of course, exactly how such locks are added determines both the correctness and performance of the data structure. And thus, our challenge:

> CRUX: HOW TO ADD LOCKS TO DATA STRUCTURES
> When given a particular data structure, how should we add locks to it, in order to make it work correctly? Further, how do we add locks such that the data structure yields high performance, enabling many threads to access the structure at once, i.e., **concurrently**?

Of course, we will be hard pressed to cover all data structures or all methods for adding concurrency, as this is a topic that has been studied for years, with (literally) thousands of research papers published about it. Thus, we hope to provide a sufficient introduction to the type of thinking required, and refer you to some good sources of material for further inquiry on your own. We found Moir and Shavit's survey to be a great source of information [MS04].

29.1 Concurrent Counters

One of the simplest data structures is a counter. It is a structure that is commonly used and has a simple interface. We define a simple non-concurrent counter in Figure 29.1.

Simple But Not Scalable

As you can see, the non-synchronized counter is a trivial data structure, requiring a tiny amount of code to implement. We now have our next challenge: how can we make this code **thread safe**? Figure 29.2 shows how we do so.

```
1  typedef struct __counter_t {
2      int value;
3  } counter_t;
4
5  void init(counter_t *c) {
6      c->value = 0;
7  }
8
9  void increment(counter_t *c) {
10     c->value++;
11 }
12
13 void decrement(counter_t *c) {
14     c->value--;
15 }
16
17 int get(counter_t *c) {
18     return c->value;
19 }
```

Figure 29.1: **A Counter Without Locks**

```
1  typedef struct __counter_t {
2      int             value;
3      pthread_mutex_t lock;
4  } counter_t;
5
6  void init(counter_t *c) {
7      c->value = 0;
8      Pthread_mutex_init(&c->lock, NULL);
9  }
10
11 void increment(counter_t *c) {
12     Pthread_mutex_lock(&c->lock);
13     c->value++;
14     Pthread_mutex_unlock(&c->lock);
15 }
16
17 void decrement(counter_t *c) {
18     Pthread_mutex_lock(&c->lock);
19     c->value--;
20     Pthread_mutex_unlock(&c->lock);
21 }
22
23 int get(counter_t *c) {
24     Pthread_mutex_lock(&c->lock);
25     int rc = c->value;
26     Pthread_mutex_unlock(&c->lock);
27     return rc;
28 }
```

Figure 29.2: **A Counter With Locks**

This concurrent counter is simple and works correctly. In fact, it follows a design pattern common to the simplest and most basic concurrent data structures: it simply adds a single lock, which is acquired when calling a routine that manipulates the data structure, and is released when returning from the call. In this manner, it is similar to a data structure built with **monitors** [BH73], where locks are acquired and released automatically as you call and return from object methods.

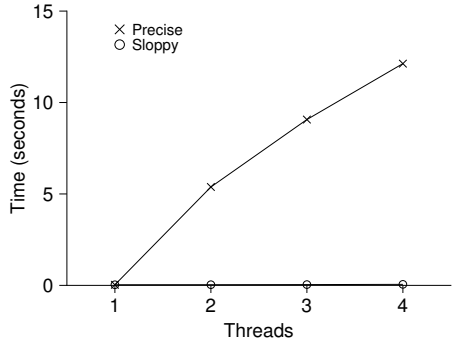


Figure 29.3: **Performance of Traditional vs. Sloppy Counters**

At this point, you have a working concurrent data structure. The problem you might have is performance. If your data structure is too slow, you'll have to do more than just add a single lock; such optimizations, if needed, are thus the topic of the rest of the chapter. Note that if the data structure is *not* too slow, you are done! No need to do something fancy if something simple will work.

To understand the performance costs of the simple approach, we run a benchmark in which each thread updates a single shared counter a fixed number of times; we then vary the number of threads. Figure 29.3 shows the total time taken, with one to four threads active; each thread updates the counter one million times. This experiment was run upon an iMac with four Intel 2.7 GHz i5 CPUs; with more CPUs active, we hope to get more total work done per unit time.

From the top line in the figure (labeled *precise*), you can see that the performance of the synchronized counter scales poorly. Whereas a single thread can complete the million counter updates in a tiny amount of time (roughly 0.03 seconds), having two threads each update the counter one million times concurrently leads to a massive slowdown (taking over 5 seconds!). It only gets worse with more threads.

Ideally, you'd like to see the threads complete just as quickly on multiple processors as the single thread does on one. Achieving this end is called **perfect scaling**; even though more work is done, it is done in parallel, and hence the time taken to complete the task is not increased.

Scalable Counting

Amazingly, researchers have studied how to build more scalable counters for years [MS04]. Even more amazing is the fact that scalable counters matter, as recent work in operating system performance analysis has shown [B+10]; without scalable counting, some workloads running on

Time	L_1	L_2	L_3	L_4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	$5 \rightarrow 0$	1	3	4	5 (from L_1)
7	0	2	4	$5 \rightarrow 0$	10 (from L_4)

Figure 29.4: **Tracing the Sloppy Counters**

Linux suffer from serious scalability problems on multicore machines.

Though many techniques have been developed to attack this problem, we'll now describe one particular approach. The idea, introduced in recent research [B+10], is known as a **sloppy counter**.

The sloppy counter works by representing a single logical counter via numerous *local* physical counters, one per CPU core, as well as a single *global* counter. Specifically, on a machine with four CPUs, there are four local counters and one global one. In addition to these counters, there are also locks: one for each local counter, and one for the global counter.

The basic idea of sloppy counting is as follows. When a thread running on a given core wishes to increment the counter, it increments its local counter; access to this local counter is synchronized via the corresponding local lock. Because each CPU has its own local counter, threads across CPUs can update local counters without contention, and thus counter updates are scalable.

However, to keep the global counter up to date (in case a thread wishes to read its value), the local values are periodically transferred to the global counter, by acquiring the global lock and incrementing it by the local counter's value; the local counter is then reset to zero.

How often this local-to-global transfer occurs is determined by a threshold, which we call S here (for sloppiness). The smaller S is, the more the counter behaves like the non-scalable counter above; the bigger S is, the more scalable the counter, but the further off the global value might be from the actual count. One could simply acquire all the local locks and the global lock (in a specified order, to avoid deadlock) to get an exact value, but that is not scalable.

To make this clear, let's look at an example (Figure 29.4). In this example, the threshold S is set to 5, and there are threads on each of four CPUs updating their local counters L_1 ... L_4. The global counter value (G) is also shown in the trace, with time increasing downward. At each time step, a local counter may be incremented; if the local value reaches the threshold S, the local value is transferred to the global counter and the local counter is reset.

The lower line in Figure 29.3 (labeled *sloppy*, on page 3) shows the performance of sloppy counters with a threshold S of 1024. Performance is excellent; the time taken to update the counter four million times on four processors is hardly higher than the time taken to update it one million times on one processor.

```
1  typedef struct __counter_t {
2      int             global;           // global count
3      pthread_mutex_t glock;            // global lock
4      int             local[NUMCPUS];   // local count (per cpu)
5      pthread_mutex_t llock[NUMCPUS];   // ... and locks
6      int             threshold;        // update frequency
7  } counter_t;
8
9  // init: record threshold, init locks, init values
10 //       of all local counts and global count
11 void init(counter_t *c, int threshold) {
12     c->threshold = threshold;
13     c->global = 0;
14     pthread_mutex_init(&c->glock, NULL);
15     int i;
16     for (i = 0; i < NUMCPUS; i++) {
17         c->local[i] = 0;
18         pthread_mutex_init(&c->llock[i], NULL);
19     }
20 }
21
22 // update: usually, just grab local lock and update local amount
23 //         once local count has risen by 'threshold', grab global
24 //         lock and transfer local values to it
25 void update(counter_t *c, int threadID, int amt) {
26     int cpu = threadID % NUMCPUS;
27     pthread_mutex_lock(&c->llock[cpu]);
28     c->local[cpu] += amt;                 // assumes amt > 0
29     if (c->local[cpu] >= c->threshold) {  // transfer to global
30         pthread_mutex_lock(&c->glock);
31         c->global += c->local[cpu];
32         pthread_mutex_unlock(&c->glock);
33         c->local[cpu] = 0;
34     }
35     pthread_mutex_unlock(&c->llock[cpu]);
36 }
37
38 // get: just return global amount (which may not be perfect)
39 int get(counter_t *c) {
40     pthread_mutex_lock(&c->glock);
41     int val = c->global;
42     pthread_mutex_unlock(&c->glock);
43     return val; // only approximate!
44 }
```

Figure 29.5: **Sloppy Counter Implementation**

Figure 29.6 shows the importance of the threshold value S, with four threads each incrementing the counter 1 million times on four CPUs. If S is low, performance is poor (but the global count is always quite accurate); if S is high, performance is excellent, but the global count lags (by at most the number of CPUs multiplied by S). This accuracy/performance trade-off is what sloppy counters enables.

A rough version of such a sloppy counter is found in Figure 29.5. Read it, or better yet, run it yourself in some experiments to better understand how it works.

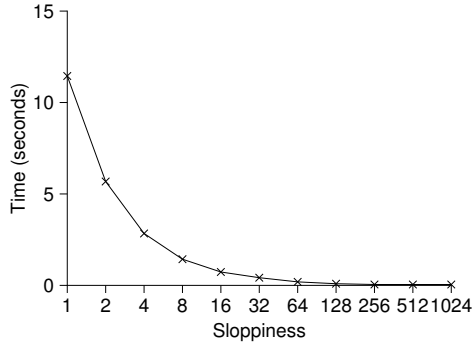


Figure 29.6: **Scaling Sloppy Counters**

29.2 Concurrent Linked Lists

We next examine a more complicated structure, the linked list. Let's start with a basic approach once again. For simplicity, we'll omit some of the obvious routines that such a list would have and just focus on concurrent insert; we'll leave it to the reader to think about lookup, delete, and so forth. Figure 29.7 shows the code for this rudimentary data structure.

As you can see in the code, the code simply acquires a lock in the insert routine upon entry, and releases it upon exit. One small tricky issue arises if `malloc()` happens to fail (a rare case); in this case, the code must also release the lock before failing the insert.

This kind of exceptional control flow has been shown to be quite error prone; a recent study of Linux kernel patches found that a huge fraction of bugs (nearly 40%) are found on such rarely-taken code paths (indeed, this observation sparked some of our own research, in which we removed all memory-failing paths from a Linux file system, resulting in a more robust system [S+11]).

Thus, a challenge: can we rewrite the insert and lookup routines to remain correct under concurrent insert but avoid the case where the failure path also requires us to add the call to unlock?

The answer, in this case, is yes. Specifically, we can rearrange the code a bit so that the lock and release only surround the actual critical section in the insert code, and that a common exit path is used in the lookup code. The former works because part of the lookup actually need not be locked; assuming that `malloc()` itself is thread-safe, each thread can call into it without worry of race conditions or other concurrency bugs. Only when updating the shared list does a lock need to be held. See Figure 29.8 for the details of these modifications.

```
1  // basic node structure
2  typedef struct __node_t {
3      int                key;
4      struct __node_t         *next;
5  } node_t;
6
7  // basic list structure (one used per list)
8  typedef struct __list_t {
9      node_t               *head;
10     pthread_mutex_t     lock;
11 } list_t;
12
13 void List_Init(list_t *L) {
14     L->head = NULL;
15     pthread_mutex_init(&L->lock, NULL);
16 }
17
18 int List_Insert(list_t *L, int key) {
19     pthread_mutex_lock(&L->lock);
20     node_t *new = malloc(sizeof(node_t));
21     if (new == NULL) {
22         perror("malloc");
23         pthread_mutex_unlock(&L->lock);
24         return -1; // fail
25     }
26     new->key  = key;
27     new->next = L->head;
28     L->head   = new;
29     pthread_mutex_unlock(&L->lock);
30     return 0; // success
31 }
32
33 int List_Lookup(list_t *L, int key) {
34     pthread_mutex_lock(&L->lock);
35     node_t *curr = L->head;
36     while (curr) {
37         if (curr->key == key) {
38             pthread_mutex_unlock(&L->lock);
39             return 0; // success
40         }
41         curr = curr->next;
42     }
43     pthread_mutex_unlock(&L->lock);
44     return -1; // failure
45 }
```

Figure 29.7: **Concurrent Linked List**

As for the lookup routine, it is a simple code transformation to jump out of the main search loop to a single return path. Doing so again reduces the number of lock acquire/release points in the code, and thus decreases the chances of accidentally introducing bugs (such as forgetting to unlock before returning) into the code.

Scaling Linked Lists

Though we again have a basic concurrent linked list, once again we are in a situation where it does not scale particularly well. One technique that researchers have explored to enable more concurrency within a list is

```
1  void List_Init(list_t *L) {
2      L->head = NULL;
3      pthread_mutex_init(&L->lock, NULL);
4  }
5
6  void List_Insert(list_t *L, int key) {
7      // synchronization not needed
8      node_t *new = malloc(sizeof(node_t));
9      if (new == NULL) {
10         perror("malloc");
11         return;
12     }
13     new->key = key;
14
15     // just lock critical section
16     pthread_mutex_lock(&L->lock);
17     new->next = L->head;
18     L->head   = new;
19     pthread_mutex_unlock(&L->lock);
20 }
21
22 int List_Lookup(list_t *L, int key) {
23     int rv = -1;
24     pthread_mutex_lock(&L->lock);
25     node_t *curr = L->head;
26     while (curr) {
27         if (curr->key == key) {
28             rv = 0;
29             break;
30         }
31         curr = curr->next;
32     }
33     pthread_mutex_unlock(&L->lock);
34     return rv; // now both success and failure
35 }
```

Figure 29.8: **Concurrent Linked List: Rewritten**

something called **hand-over-hand locking** (a.k.a. **lock coupling**) [MS04].

The idea is pretty simple. Instead of having a single lock for the entire list, you instead add a lock per node of the list. When traversing the list, the code first grabs the next node's lock and then releases the current node's lock (which inspires the name hand-over-hand).

Conceptually, a hand-over-hand linked list makes some sense; it enables a high degree of concurrency in list operations. However, in practice, it is hard to make such a structure faster than the simple single lock approach, as the overheads of acquiring and releasing locks for each node of a list traversal is prohibitive. Even with very large lists, and a large number of threads, the concurrency enabled by allowing multiple ongoing traversals is unlikely to be faster than simply grabbing a single lock, performing an operation, and releasing it. Perhaps some kind of hybrid (where you grab a new lock every so many nodes) would be worth investigating.

TIP: MORE CONCURRENCY ISN'T NECESSARILY FASTER
If the scheme you design adds a lot of overhead (for example, by acquiring and releasing locks frequently, instead of once), the fact that it is more concurrent may not be important. Simple schemes tend to work well, especially if they use costly routines rarely. Adding more locks and complexity can be your downfall. All of that said, there is one way to really know: build both alternatives (simple but less concurrent, and complex but more concurrent) and measure how they do. In the end, you can't cheat on performance; your idea is either faster, or it isn't.

TIP: BE WARY OF LOCKS AND CONTROL FLOW
A general design tip, which is useful in concurrent code as well as elsewhere, is to be wary of control flow changes that lead to function returns, exits, or other similar error conditions that halt the execution of a function. Because many functions will begin by acquiring a lock, allocating some memory, or doing other similar stateful operations, when errors arise, the code has to undo all of the state before returning, which is error-prone. Thus, it is best to structure code to minimize this pattern.

29.3 Concurrent Queues

As you know by now, there is always a standard method to make a concurrent data structure: add a big lock. For a queue, we'll skip that approach, assuming you can figure it out.

Instead, we'll take a look at a slightly more concurrent queue designed by Michael and Scott [MS98]. The data structures and code used for this queue are found in Figure 29.9 on the following page.

If you study this code carefully, you'll notice that there are two locks, one for the head of the queue, and one for the tail. The goal of these two locks is to enable concurrency of enqueue and dequeue operations. In the common case, the enqueue routine will only access the tail lock, and dequeue only the head lock.

One trick used by Michael and Scott is to add a dummy node (allocated in the queue initialization code); this dummy enables the separation of head and tail operations. Study the code, or better yet, type it in, run it, and measure it, to understand how it works deeply.

Queues are commonly used in multi-threaded applications. However, the type of queue used here (with just locks) often does not completely meet the needs of such programs. A more fully developed bounded queue, that enables a thread to wait if the queue is either empty or overly full, is the subject of our intense study in the next chapter on condition variables. Watch for it!

```
1  typedef struct __node_t {
2      int                 value;
3      struct __node_t    *next;
4  } node_t;
5
6  typedef struct __queue_t {
7      node_t             *head;
8      node_t             *tail;
9      pthread_mutex_t     headLock;
10     pthread_mutex_t     tailLock;
11 } queue_t;
12
13 void Queue_Init(queue_t *q) {
14     node_t *tmp = malloc(sizeof(node_t));
15     tmp->next = NULL;
16     q->head = q->tail = tmp;
17     pthread_mutex_init(&q->headLock, NULL);
18     pthread_mutex_init(&q->tailLock, NULL);
19 }
20
21 void Queue_Enqueue(queue_t *q, int value) {
22     node_t *tmp = malloc(sizeof(node_t));
23     assert(tmp != NULL);
24     tmp->value = value;
25     tmp->next  = NULL;
26
27     pthread_mutex_lock(&q->tailLock);
28     q->tail->next = tmp;
29     q->tail = tmp;
30     pthread_mutex_unlock(&q->tailLock);
31 }
32
33 int Queue_Dequeue(queue_t *q, int *value) {
34     pthread_mutex_lock(&q->headLock);
35     node_t *tmp = q->head;
36     node_t *newHead = tmp->next;
37     if (newHead == NULL) {
38         pthread_mutex_unlock(&q->headLock);
39         return -1; // queue was empty
40     }
41     *value = newHead->value;
42     q->head = newHead;
43     pthread_mutex_unlock(&q->headLock);
44     free(tmp);
45     return 0;
46 }
```

Figure 29.9: **Michael and Scott Concurrent Queue**

29.4 Concurrent Hash Table

We end our discussion with a simple and widely applicable concurrent data structure, the hash table. We'll focus on a simple hash table that does not resize; a little more work is required to handle resizing, which we leave as an exercise for the reader (sorry!).

This concurrent hash table is straightforward, is built using the concurrent lists we developed earlier, and works incredibly well. The reason

```
1  #define BUCKETS (101)
2
3  typedef struct __hash_t {
4      list_t lists[BUCKETS];
5  } hash_t;
6
7  void Hash_Init(hash_t *H) {
8      int i;
9      for (i = 0; i < BUCKETS; i++) {
10         List_Init(&H->lists[i]);
11     }
12 }
13
14 int Hash_Insert(hash_t *H, int key) {
15     int bucket = key % BUCKETS;
16     return List_Insert(&H->lists[bucket], key);
17 }
18
19 int Hash_Lookup(hash_t *H, int key) {
20     int bucket = key % BUCKETS;
21     return List_Lookup(&H->lists[bucket], key);
22 }
```

Figure 29.10: **A Concurrent Hash Table**

for its good performance is that instead of having a single lock for the entire structure, it uses a lock per hash bucket (each of which is represented by a list). Doing so enables many concurrent operations to take place.

Figure 29.11 shows the performance of the hash table under concurrent updates (from 10,000 to 50,000 concurrent updates from each of four threads, on the same iMac with four CPUs). Also shown, for the sake of comparison, is the performance of a linked list (with a single lock). As you can see from the graph, this simple concurrent hash table scales magnificently; the linked list, in contrast, does not.

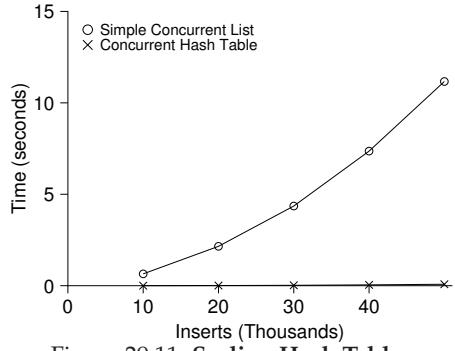


Figure 29.11: **Scaling Hash Tables**

TIP: AVOID PREMATURE OPTIMIZATION (KNUTH'S LAW)

When building a concurrent data structure, start with the most basic approach, which is to add a single big lock to provide synchronized access. By doing so, you are likely to build a *correct* lock; if you then find that it suffers from performance problems, you can refine it, thus only making it fast if need be. As **Knuth** famously stated, "Premature optimization is the root of all evil."

Many operating systems utilized a single lock when first transitioning to multiprocessors, including Sun OS and Linux. In the latter, this lock even had a name, the **big kernel lock** (**BKL**). For many years, this simple approach was a good one, but when multi-CPU systems became the norm, only allowing a single active thread in the kernel at a time became a performance bottleneck. Thus, it was finally time to add the optimization of improved concurrency to these systems. Within Linux, the more straightforward approach was taken: replace one lock with many. Within Sun, a more radical decision was made: build a brand new operating system, known as Solaris, that incorporates concurrency more fundamentally from day one. Read the Linux and Solaris kernel books for more information about these fascinating systems [BC05, MM00].

29.5 Summary

We have introduced a sampling of concurrent data structures, from counters, to lists and queues, and finally to the ubiquitous and heavily-used hash table. We have learned a few important lessons along the way: to be careful with acquisition and release of locks around control flow changes; that enabling more concurrency does not necessarily increase performance; that performance problems should only be remedied once they exist. This last point, of avoiding **premature optimization**, is central to any performance-minded developer; there is no value in making something faster if doing so will not improve the overall performance of the application.

Of course, we have just scratched the surface of high performance structures. See Moir and Shavit's excellent survey for more information, as well as links to other sources [MS04]. In particular, you might be interested in other structures (such as B-trees); for this knowledge, a database class is your best bet. You also might be interested in techniques that don't use traditional locks at all; such **non-blocking data structures** are something we'll get a taste of in the chapter on common concurrency bugs, but frankly this topic is an entire area of knowledge requiring more study than is possible in this humble book. Find out more on your own if you are interested (as always!).

References

[B+10] "An Analysis of Linux Scalability to Many Cores"
Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, Nickolai Zeldovich
OSDI '10, Vancouver, Canada, October 2010
A great study of how Linux performs on multicore machines, as well as some simple solutions.

[BH73] "Operating System Principles"
Per Brinch Hansen, Prentice-Hall, 1973
Available: http://portal.acm.org/citation.cfm?id=540365
One of the first books on operating systems; certainly ahead of its time. Introduced monitors as a concurrency primitive.

[BC05] "Understanding the Linux Kernel (Third Edition)"
Daniel P. Bovet and Marco Cesati
O'Reilly Media, November 2005
The classic book on the Linux kernel. You should read it.

[L+13] "A Study of Linux File System Evolution"
Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Shan Lu
FAST '13, San Jose, CA, February 2013
Our paper that studies every patch to Linux file systems over nearly a decade. Lots of fun findings in there; read it to see! The work was painful to do though; the poor graduate student, Lanyue Lu, had to look through every single patch by hand in order to understand what they did.

[MS98] "Nonblocking Algorithms and Preemption-safe Locking on Multiprogrammed Shared-memory Multiprocessors"
M. Michael and M. Scott
Journal of Parallel and Distributed Computing, Vol. 51, No. 1, 1998
Professor Scott and his students have been at the forefront of concurrent algorithms and data structures for many years; check out his web page, numerous papers, or books to find out more.

[MS04] "Concurrent Data Structures"
Mark Moir and Nir Shavit
In Handbook of Data Structures and Applications
(Editors D. Metha and S.Sahni)
Chapman and Hall/CRC Press, 2004
Available: www.cs.tau.ac.il/~shanir/concurrent-data-structures.pdf
A short but relatively comprehensive reference on concurrent data structures. Though it is missing some of the latest works in the area (due to its age), it remains an incredibly useful reference.

[MM00] "Solaris Internals: Core Kernel Architecture"
Jim Mauro and Richard McDougall
Prentice Hall, October 2000
The Solaris book. You should also read this, if you want to learn in great detail about something other than Linux.

[S+11] "Making the Common Case the Only Case with Anticipatory Memory Allocation"
Swaminathan Sundararaman, Yupu Zhang, Sriram Subramanian,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
FAST '11, San Jose, CA, February 2011
Our work on removing possibly-failing calls to malloc from kernel code paths. The idea is to allocate all potentially needed memory before doing any of the work, thus avoiding failure deep down in the storage stack.

30

Condition Variables

Thus far we have developed the notion of a lock and seen how one can be properly built with the right combination of hardware and OS support. Unfortunately, locks are not the only primitives that are needed to build concurrent programs.

In particular, there are many cases where a thread wishes to check whether a **condition** is true before continuing its execution. For example, a parent thread might wish to check whether a child thread has completed before continuing (this is often called a `join()`); how should such a wait be implemented? Let's look at Figure 30.1.

```
1  void *child(void *arg) {
2      printf("child\n");
3      // XXX how to indicate we are done?
4      return NULL;
5  }
6
7  int main(int argc, char *argv[]) {
8      printf("parent: begin\n");
9      pthread_t c;
10     Pthread_create(&c, NULL, child, NULL); // create child
11     // XXX how to wait for child?
12     printf("parent: end\n");
13     return 0;
14 }
```

Figure 30.1: **A Parent Waiting For Its Child**

What we would like to see here is the following output:

```
parent: begin
child
parent: end
```

We could try using a shared variable, as you see in Figure 30.2. This solution will generally work, but it is hugely inefficient as the parent spins and wastes CPU time. What we would like here instead is some way to put the parent to sleep until the condition we are waiting for (e.g., the child is done executing) comes true.

```
1  volatile int done = 0;
2
3  void *child(void *arg) {
4      printf("child\n");
5      done = 1;
6      return NULL;
7  }
8
9  int main(int argc, char *argv[]) {
10     printf("parent: begin\n");
11     pthread_t c;
12     Pthread_create(&c, NULL, child, NULL); // create child
13     while (done == 0)
14         ; // spin
15     printf("parent: end\n");
16     return 0;
17 }
```

Figure 30.2: **Parent Waiting For Child: Spin-based Approach**

THE CRUX: HOW TO WAIT FOR A CONDITION

In multi-threaded programs, it is often useful for a thread to wait for some condition to become true before proceeding. The simple approach, of just spinning until the condition becomes true, is grossly inefficient and wastes CPU cycles, and in some cases, can be incorrect. Thus, how should a thread wait for a condition?

30.1 Definition and Routines

To wait for a condition to become true, a thread can make use of what is known as a **condition variable**. A **condition variable** is an explicit queue that threads can put themselves on when some state of execution (i.e., some **condition**) is not as desired (by **waiting** on the condition); some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue (by **signaling** on the condition). The idea goes back to Dijkstra's use of "private semaphores" [D68]; a similar idea was later named a "condition variable" by Hoare in his work on monitors [H74].

To declare such a condition variable, one simply writes something like this: `pthread_cond_t c;`, which declares `c` as a condition variable (note: proper initialization is also required). A condition variable has two operations associated with it: `wait()` and `signal()`. The `wait()` call is executed when a thread wishes to put itself to sleep; the `signal()` call is executed when a thread has changed something in the program and thus wants to wake a sleeping thread waiting on this condition. Specifically, the POSIX calls look like this:

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

```
1  int done  = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

Figure 30.3: **Parent Waiting For Child: Use A Condition Variable**

We will often refer to these as `wait()` and `signal()` for simplicity. One thing you might notice about the `wait()` call is that it also takes a mutex as a parameter; it assumes that this mutex is locked when `wait()` is called. The responsibility of `wait()` is to release the lock and put the calling thread to sleep (atomically); when the thread wakes up (after some other thread has signaled it), it must re-acquire the lock before returning to the caller. This complexity stems from the desire to prevent certain race conditions from occurring when a thread is trying to put itself to sleep. Let's take a look at the solution to the join problem (Figure 30.3) to understand this better.

There are two cases to consider. In the first, the parent creates the child thread but continues running itself (assume we have only a single processor) and thus immediately calls into `thr_join()` to wait for the child thread to complete. In this case, it will acquire the lock, check if the child is done (it is not), and put itself to sleep by calling `wait()` (hence releasing the lock). The child will eventually run, print the message "child", and call `thr_exit()` to wake the parent thread; this code just grabs the lock, sets the state variable `done`, and signals the parent thus waking it. Finally, the parent will run (returning from `wait()` with the lock held), unlock the lock, and print the final message "parent: end".

In the second case, the child runs immediately upon creation, sets `done` to 1, calls signal to wake a sleeping thread (but there is none, so it just returns), and is done. The parent then runs, calls `thr_join()`, sees that `done` is 1, and thus does not wait and returns.

One last note: you might observe the parent uses a `while` loop instead of just an `if` statement when deciding whether to wait on the condition. While this does not seem strictly necessary per the logic of the program, it is always a good idea, as we will see below.

To make sure you understand the importance of each piece of the `thr_exit()` and `thr_join()` code, let's try a few alternate implementations. First, you might be wondering if we need the state variable `done`. What if the code looked like the example below? Would this work?

```
1  void thr_exit() {
2      Pthread_mutex_lock(&m);
3      Pthread_cond_signal(&c);
4      Pthread_mutex_unlock(&m);
5  }
6
7  void thr_join() {
8      Pthread_mutex_lock(&m);
9      Pthread_cond_wait(&c, &m);
10     Pthread_mutex_unlock(&m);
11 }
```

Unfortunately this approach is broken. Imagine the case where the child runs immediately and calls `thr_exit()` immediately; in this case, the child will signal, but there is no thread asleep on the condition. When the parent runs, it will simply call wait and be stuck; no thread will ever wake it. From this example, you should appreciate the importance of the state variable `done`; it records the value the threads are interested in knowing. The sleeping, waking, and locking all are built around it.

Here is another poor implementation. In this example, we imagine that one does not need to hold a lock in order to signal and wait. What problem could occur here? Think about it!

```
1  void thr_exit() {
2      done = 1;
3      Pthread_cond_signal(&c);
4  }
5
6  void thr_join() {
7      if (done == 0)
8          Pthread_cond_wait(&c);
9  }
```

The issue here is a subtle race condition. Specifically, if the parent calls `thr_join()` and then checks the value of `done`, it will see that it is 0 and thus try to go to sleep. But just before it calls wait to go to sleep, the parent is interrupted, and the child runs. The child changes the state variable `done` to 1 and signals, but no thread is waiting and thus no thread is woken. When the parent runs again, it sleeps forever, which is sad.

TIP: ALWAYS HOLD THE LOCK WHILE SIGNALING

Although it is strictly not necessary in all cases, it is likely simplest and best to hold the lock while signaling when using condition variables. The example above shows a case where you *must* hold the lock for correctness; however, there are some other cases where it is likely OK not to, but probably is something you should avoid. Thus, for simplicity, **hold the lock when calling signal**.

The converse of this tip, i.e., hold the lock when calling wait, is not just a tip, but rather mandated by the semantics of wait, because wait always (a) assumes the lock is held when you call it, (b) releases said lock when putting the caller to sleep, and (c) re-acquires the lock just before returning. Thus, the generalization of this tip is correct: **hold the lock when calling signal or wait**, and you will always be in good shape.

Hopefully, from this simple join example, you can see some of the basic requirements of using condition variables properly. To make sure you understand, we now go through a more complicated example: the **producer/consumer** or **bounded-buffer** problem.

30.2 The Producer/Consumer (Bounded Buffer) Problem

The next synchronization problem we will confront in this chapter is known as the **producer/consumer** problem, or sometimes as the **bounded buffer** problem, which was first posed by Dijkstra [D72]. Indeed, it was this very producer/consumer problem that led Dijkstra and his co-workers to invent the generalized semaphore (which can be used as either a lock or a condition variable) [D01]; we will learn more about semaphores later.

Imagine one or more producer threads and one or more consumer threads. Producers generate data items and place them in a buffer; consumers grab said items from the buffer and consume them in some way.

This arrangement occurs in many real systems. For example, in a multi-threaded web server, a producer puts HTTP requests into a work queue (i.e., the bounded buffer); consumer threads take requests out of this queue and process them.

A bounded buffer is also used when you pipe the output of one program into another, e.g., `grep foo file.txt | wc -l`. This example runs two processes concurrently; `grep` writes lines from `file.txt` with the string `foo` in them to what it thinks is standard output; the UNIX shell redirects the output to what is called a UNIX pipe (created by the **pipe** system call). The other end of this pipe is connected to the standard input of the process `wc`, which simply counts the number of lines in the input stream and prints out the result. Thus, the `grep` process is the producer; the `wc` process is the consumer; between them is an in-kernel bounded buffer; you, in this example, are just the happy user.

```
1  int buffer;
2  int count = 0; // initially, empty
3
4  void put(int value) {
5      assert(count == 0);
6      count = 1;
7      buffer = value;
8  }
9
10 int get() {
11     assert(count == 1);
12     count = 0;
13     return buffer;
14 }
```

Figure 30.4: **The Put And Get Routines (Version 1)**

```
1  void *producer(void *arg) {
2      int i;
3      int loops = (int) arg;
4      for (i = 0; i < loops; i++) {
5          put(i);
6      }
7  }
8
9  void *consumer(void *arg) {
10     int i;
11     while (1) {
12         int tmp = get();
13         printf("%d\n", tmp);
14     }
15 }
```

Figure 30.5: **Producer/Consumer Threads (Version 1)**

Because the bounded buffer is a shared resource, we must of course require synchronized access to it, lest[1] a race condition arise. To begin to understand this problem better, let us examine some actual code.

The first thing we need is a shared buffer, into which a producer puts data, and out of which a consumer takes data. Let's just use a single integer for simplicity (you can certainly imagine placing a pointer to a data structure into this slot instead), and the two inner routines to put a value into the shared buffer, and to get a value out of the buffer. See Figure 30.4 for details.

Pretty simple, no? The `put()` routine assumes the buffer is empty (and checks this with an assertion), and then simply puts a value into the shared buffer and marks it full by setting `count` to 1. The `get()` routine does the opposite, setting the buffer to empty (i.e., setting `count` to 0) and returning the value. Don't worry that this shared buffer has just a single entry; later, we'll generalize it to a queue that can hold multiple entries, which will be even more fun than it sounds.

Now we need to write some routines that know when it is OK to access the buffer to either put data into it or get data out of it. The conditions for

[1] This is where we drop some serious Old English on you, and the subjunctive form.

```
1  cond_t  cond;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);            // p1
8          if (count == 1)                        // p2
9              Pthread_cond_wait(&cond, &mutex); // p3
10         put(i);                                // p4
11         Pthread_cond_signal(&cond);            // p5
12         Pthread_mutex_unlock(&mutex);          // p6
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);            // c1
20         if (count == 0)                        // c2
21             Pthread_cond_wait(&cond, &mutex); // c3
22         int tmp = get();                       // c4
23         Pthread_cond_signal(&cond);            // c5
24         Pthread_mutex_unlock(&mutex);          // c6
25         printf("%d\n", tmp);
26     }
27 }
```

Figure 30.6: **Producer/Consumer: Single CV And If Statement**

this should be obvious: only put data into the buffer when `count` is zero (i.e., when the buffer is empty), and only get data from the buffer when `count` is one (i.e., when the buffer is full). If we write the synchronization code such that a producer puts data into a full buffer, or a consumer gets data from an empty one, we have done something wrong (and in this code, an assertion will fire).

This work is going to be done by two types of threads, one set of which we'll call the **producer** threads, and the other set which we'll call **consumer** threads. Figure 30.5 shows the code for a producer that puts an integer into the shared buffer `loops` number of times, and a consumer that gets the data out of that shared buffer (forever), each time printing out the data item it pulled from the shared buffer.

A Broken Solution

Now imagine that we have just a single producer and a single consumer. Obviously the `put()` and `get()` routines have critical sections within them, as `put()` updates the buffer, and `get()` reads from it. However, putting a lock around the code doesn't work; we need something more. Not surprisingly, that something more is some condition variables. In this (broken) first try (Figure 30.6), we have a single condition variable `cond` and associated lock `mutex`.

T_{c1}	State	T_{c2}	State	T_p	State	**Count**	*Comment*
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	Buffer now full
	Ready		Ready	p5	Running	1	T_{c1} awoken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Running		Sleep	1	T_{c2} sneaks in ...
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	... and grabs data
	Ready	c5	Running		Ready	0	T_p awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	Oh oh! No data

Figure 30.7: **Thread Trace: Broken Solution (Version 1)**

Let's examine the signaling logic between producers and consumers. When a producer wants to fill the buffer, it waits for it to be empty (p1–p3). The consumer has the exact same logic, but waits for a different condition: fullness (c1–c3).

With just a single producer and a single consumer, the code in Figure 30.6 works. However, if we have more than one of these threads (e.g., two consumers), the solution has two critical problems. What are they?

... (pause here to think) ...

Let's understand the first problem, which has to do with the `if` statement before the wait. Assume there are two consumers (T_{c1} and T_{c2}) and one producer (T_p). First, a consumer (T_{c1}) runs; it acquires the lock (c1), checks if any buffers are ready for consumption (c2), and finding that none are, waits (c3) (which releases the lock).

Then the producer (T_p) runs. It acquires the lock (p1), checks if all buffers are full (p2), and finding that not to be the case, goes ahead and fills the buffer (p4). The producer then signals that a buffer has been filled (p5). Critically, this moves the first consumer (T_{c1}) from sleeping on a condition variable to the ready queue; T_{c1} is now able to run (but not yet running). The producer then continues until realizing the buffer is full, at which point it sleeps (p6, p1–p3).

Here is where the problem occurs: another consumer (T_{c2}) sneaks in and consumes the one existing value in the buffer (c1, c2, c4, c5, c6, skipping the wait at c3 because the buffer is full). Now assume T_{c1} runs; just before returning from the wait, it re-acquires the lock and then returns. It then calls `get()` (c4), but there are no buffers to consume! An assertion triggers, and the code has not functioned as desired. Clearly, we should have somehow prevented T_{c1} from trying to consume because T_{c2} snuck in and consumed the one value in the buffer that had been produced. Figure 30.7 shows the action each thread takes, as well as its scheduler state (Ready, Running, or Sleeping) over time.

```
1  cond_t  cond;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);            // p1
8          while (count == 1)                     // p2
9              Pthread_cond_wait(&cond, &mutex); // p3
10         put(i);                                // p4
11         Pthread_cond_signal(&cond);            // p5
12         Pthread_mutex_unlock(&mutex);          // p6
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);            // c1
20         while (count == 0)                     // c2
21             Pthread_cond_wait(&cond, &mutex); // c3
22         int tmp = get();                       // c4
23         Pthread_cond_signal(&cond);            // c5
24         Pthread_mutex_unlock(&mutex);          // c6
25         printf("%d\n", tmp);
26     }
27 }
```

Figure 30.8: **Producer/Consumer: Single CV And While**

The problem arises for a simple reason: after the producer woke T_{c1}, but *before* T_{c1} ever ran, the state of the bounded buffer changed (thanks to T_{c2}). Signaling a thread only wakes them up; it is thus a *hint* that the state of the world has changed (in this case, that a value has been placed in the buffer), but there is no guarantee that when the woken thread runs, the state will *still* be as desired. This interpretation of what a signal means is often referred to as **Mesa semantics**, after the first research that built a condition variable in such a manner [LR80]; the contrast, referred to as **Hoare semantics**, is harder to build but provides a stronger guarantee that the woken thread will run immediately upon being woken [H74]. Virtually every system ever built employs Mesa semantics.

Better, But Still Broken: While, Not If

Fortunately, this fix is easy (Figure 30.8): change the `if` to a `while`. Think about why this works; now consumer T_{c1} wakes up and (with the lock held) immediately re-checks the state of the shared variable (c2). If the buffer is empty at that point, the consumer simply goes back to sleep (c3). The corollary `if` is also changed to a `while` in the producer (p2).

Thanks to Mesa semantics, a simple rule to remember with condition variables is to **always use while loops**. Sometimes you don't have to re-check the condition, but it is always safe to do so; just do it and be happy.

T_{c1}	State	T_{c2}	State	T_p	State	**Count**	*Comment*
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	T_{c1} awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Running		Sleep		Sleep	1	Recheck condition
c4	Running		Sleep		Sleep	0	T_{c1} grabs data
c5	Running		Ready		Sleep	0	Oops! Woke T_{c2}
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Running		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep...

Figure 30.9: **Thread Trace: Broken Solution (Version 2)**

However, this code still has a bug, the second of two problems mentioned above. Can you see it? It has something to do with the fact that there is only one condition variable. Try to figure out what the problem is, before reading ahead. DO IT!

... (another pause for you to think, or close your eyes for a bit) ...

Let's confirm you figured it out correctly, or perhaps let's confirm that you are now awake and reading this part of the book. The problem occurs when two consumers run first (T_{c1} and T_{c2}), and both go to sleep (c3). Then, a producer runs, put a value in the buffer, wakes one of the consumers (say T_{c1}), and goes back to sleep. Now we have one consumer ready to run (T_{c1}), and two threads sleeping on a condition (T_{c2} and T_p). And we are about to cause a problem to occur: things are getting exciting!

The consumer T_{c1} then wakes by returning from `wait()` (c3), re-checks the condition (c2), and finding the buffer full, consumes the value (c4). This consumer then, critically, signals on the condition (c5), waking one thread that is sleeping. However, which thread should it wake?

Because the consumer has emptied the buffer, it clearly should wake the producer. However, if it wakes the consumer T_{c2} (which is definitely possible, depending on how the wait queue is managed), we have a problem. Specifically, the consumer T_{c2} will wake up and find the buffer empty (c2), and go back to sleep (c3). The producer T_p, which has a value to put into the buffer, is left sleeping. The other consumer thread, T_{c1}, also goes back to sleep. All three threads are left sleeping, a clear bug; see Figure 30.9 for the brutal step-by-step of this terrible calamity.

Signaling is clearly needed, but must be more directed. A consumer should not wake other consumers, only producers, and vice-versa.

```
1  cond_t  empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);
8          while (count == 1)
9              Pthread_cond_wait(&empty, &mutex);
10         put(i);
11         Pthread_cond_signal(&fill);
12         Pthread_mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);
20         while (count == 0)
21             Pthread_cond_wait(&fill, &mutex);
22         int tmp = get();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

Figure 30.10: **Producer/Consumer: Two CVs And While**

The Single Buffer Producer/Consumer Solution

The solution here is once again a small one: use *two* condition variables, instead of one, in order to properly signal which type of thread should wake up when the state of the system changes. Figure 30.10 shows the resulting code.

In the code above, producer threads wait on the condition **empty**, and signals **fill**. Conversely, consumer threads wait on **fill** and signal **empty**. By doing so, the second problem above is avoided by design: a consumer can never accidentally wake a consumer, and a producer can never accidentally wake a producer.

The Final Producer/Consumer Solution

We now have a working producer/consumer solution, albeit not a fully general one. The last change we make is to enable more concurrency and efficiency; specifically, we add more buffer slots, so that multiple values can be produced before sleeping, and similarly multiple values can be consumed before sleeping. With just a single producer and consumer, this approach is more efficient as it reduces context switches; with multiple producers or consumers (or both), it even allows concurrent producing or consuming to take place, thus increasing concurrency. Fortunately, it is a small change from our current solution.

```
1  int buffer[MAX];
2  int fill_ptr = 0;
3  int use_ptr  = 0;
4  int count    = 0;
5
6  void put(int value) {
7      buffer[fill_ptr] = value;
8      fill_ptr = (fill_ptr + 1) % MAX;
9      count++;
10 }
11
12 int get() {
13     int tmp = buffer[use_ptr];
14     use_ptr = (use_ptr + 1) % MAX;
15     count--;
16     return tmp;
17 }
```

Figure 30.11: **The Final Put And Get Routines**

```
1  cond_t empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);            // p1
8          while (count == MAX)                   // p2
9              Pthread_cond_wait(&empty, &mutex); // p3
10         put(i);                                // p4
11         Pthread_cond_signal(&fill);            // p5
12         Pthread_mutex_unlock(&mutex);          // p6
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);            // c1
20         while (count == 0)                     // c2
21             Pthread_cond_wait(&fill, &mutex);  // c3
22         int tmp = get();                       // c4
23         Pthread_cond_signal(&empty);           // c5
24         Pthread_mutex_unlock(&mutex);          // c6
25         printf("%d\n", tmp);
26     }
27 }
```

Figure 30.12: **The Final Working Solution**

The first change for this final solution is within the buffer structure itself and the corresponding `put()` and `get()` (Figure 30.11). We also slightly change the conditions that producers and consumers check in order to determine whether to sleep or not. Figure 30.12 shows the final waiting and signaling logic. A producer only sleeps if all buffers are currently filled (p2); similarly, a consumer only sleeps if all buffers are currently empty (c2). And thus we solve the producer/consumer problem.

TIP: USE WHILE (NOT IF) FOR CONDITIONS

When checking for a condition in a multi-threaded program, using a `while` loop is always correct; using an `if` statement only might be, depending on the semantics of signaling. Thus, always use `while` and your code will behave as expected.

Using while loops around conditional checks also handles the case where **spurious wakeups** occur. In some thread packages, due to details of the implementation, it is possible that two threads get woken up though just a single signal has taken place [L11]. Spurious wakeups are further reason to re-check the condition a thread is waiting on.

30.3 Covering Conditions

We'll now look at one more example of how condition variables can be used. This code study is drawn from Lampson and Redell's paper on Pilot [LR80], the same group who first implemented the **Mesa semantics** described above (the language they used was Mesa, hence the name).

The problem they ran into is best shown via simple example, in this case in a simple multi-threaded memory allocation library. Figure 30.13 shows a code snippet which demonstrates the issue.

As you might see in the code, when a thread calls into the memory allocation code, it might have to wait in order for more memory to become free. Conversely, when a thread frees memory, it signals that more memory is free. However, our code above has a problem: which waiting thread (there can be more than one) should be woken up?

Consider the following scenario. Assume there are zero bytes free; thread T_a calls `allocate(100)`, followed by thread T_b which asks for less memory by calling `allocate(10)`. Both T_a and T_b thus wait on the condition and go to sleep; there aren't enough free bytes to satisfy either of these requests.

At that point, assume a third thread, T_c, calls `free(50)`. Unfortunately, when it calls signal to wake a waiting thread, it might not wake the correct waiting thread, T_b, which is waiting for only 10 bytes to be freed; T_a should remain waiting, as not enough memory is yet free. Thus, the code in the figure does not work, as the thread waking other threads does not know which thread (or threads) to wake up.

The solution suggested by Lampson and Redell is straightforward: replace the `pthread_cond_signal()` call in the code above with a call to `pthread_cond_broadcast()`, which wakes up *all* waiting threads. By doing so, we guarantee that any threads that should be woken are. The downside, of course, can be a negative performance impact, as we might needlessly wake up many other waiting threads that shouldn't (yet) be awake. Those threads will simply wake up, re-check the condition, and then go immediately back to sleep.

```
1  // how many bytes of the heap are free?
2  int bytesLeft = MAX_HEAP_SIZE;
3
4  // need lock and condition too
5  cond_t  c;
6  mutex_t m;
7
8  void *
9  allocate(int size) {
10     Pthread_mutex_lock(&m);
11     while (bytesLeft < size)
12         Pthread_cond_wait(&c, &m);
13     void *ptr = ...; // get mem from heap
14     bytesLeft -= size;
15     Pthread_mutex_unlock(&m);
16     return ptr;
17 }
18
19 void free(void *ptr, int size) {
20     Pthread_mutex_lock(&m);
21     bytesLeft += size;
22     Pthread_cond_signal(&c); // whom to signal??
23     Pthread_mutex_unlock(&m);
24 }
```

Figure 30.13: **Covering Conditions: An Example**

Lampson and Redell call such a condition a **covering condition**, as it covers all the cases where a thread needs to wake up (conservatively); the cost, as we've discussed, is that too many threads might be woken. The astute reader might also have noticed we could have used this approach earlier (see the producer/consumer problem with only a single condition variable). However, in that case, a better solution was available to us, and thus we used it. In general, if you find that your program only works when you change your signals to broadcasts (but you don't think it should need to), you probably have a bug; fix it! But in cases like the memory allocator above, broadcast may be the most straightforward solution available.

30.4 Summary

We have seen the introduction of another important synchronization primitive beyond locks: condition variables. By allowing threads to sleep when some program state is not as desired, CVs enable us to neatly solve a number of important synchronization problems, including the famous (and still important) producer/consumer problem, as well as covering conditions. A more dramatic concluding sentence would go here, such as "He loved Big Brother" [O49].

References

[D68] "Cooperating sequential processes"
Edsger W. Dijkstra, 1968
Available: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF
Another classic from Dijkstra; reading his early works on concurrency will teach you much of what you need to know.

[D72] "Information Streams Sharing a Finite Buffer"
E.W. Dijkstra
Information Processing Letters 1: 179180, 1972
Available: http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD329.PDF
The famous paper that introduced the producer/consumer problem.

[D01] "My recollections of operating system design"
E.W. Dijkstra
April, 2001
Available: http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1303.PDF
A fascinating read for those of you interested in how the pioneers of our field came up with some very basic and fundamental concepts, including ideas like "interrupts" and even "a stack"!

[H74] "Monitors: An Operating System Structuring Concept"
C.A.R. Hoare
Communications of the ACM, 17:10, pages 549–557, October 1974
Hoare did a fair amount of theoretical work in concurrency. However, he is still probably most known for his work on Quicksort, the coolest sorting algorithm in the world, at least according to these authors.

[L11] "Pthread_cond_signal Man Page"
Available: http://linux.die.net/man/3/pthread_cond_signal
March, 2011
The Linux man page shows a nice simple example of why a thread might get a spurious wakeup, due to race conditions within the signal/wakeup code.

[LR80] "Experience with Processes and Monitors in Mesa"
B.W. Lampson, D.R. Redell
Communications of the ACM. 23:2, pages 105-117, February 1980
A terrific paper about how to actually implement signaling and condition variables in a real system, leading to the term "Mesa" semantics for what it means to be woken up; the older semantics, developed by Tony Hoare [H74], then became known as "Hoare" semantics, which is hard to say out loud in class with a straight face.

[O49] "1984"
George Orwell, 1949, Secker and Warburg
A little heavy-handed, but of course a must read. That said, we kind of gave away the ending by quoting the last sentence. Sorry! And if the government is reading this, let us just say that we think that the government is "double plus good". Hear that, our pals at the NSA?

Homework

This homework lets you explore some real code that uses locks and condition variables to implement various forms of the producer/consumer queue discussed in the chapter. You'll look at the real code, run it in various configurations, and use it to learn about what works and what doesn't, as well as other intricacies.

The different versions of the code correspond to different ways to "solve" the producer/consumer problem. Most are incorrect; one is correct. Read the chapter to learn more about what the producer/consumer problem is, and what the code generally does.

The first step is to download the code and type `make` to build all the variants. You should see four:

- `main-one-cv-while.c`: The producer/consumer problem solved with a single condition variable.
- `main-two-cvs-if.c`: Same but with two condition variables and using an `if` to check whether to sleep.
- `main-two-cvs-while.c`: Same but with two condition variables and `while` to check whether to sleep. **This is the correct version.**
- `main-two-cvs-while-extra-unlock.c`: Same but releasing the lock and then reacquiring it around the fill and get routines.

It's also useful to look at `pc-header.h` which contains common code for all of these different main programs, and the `Makefile` so as to build the code properly.

See the README for details on these programs.

Questions

1. Our first question focuses on `main-two-cvs-while.c` (the working solution). First, study the code. Do you think you have an understanding of what should happen when you run the program?
2. Now run with one producer and one consumer, and have the producer produce a few values. Start with a buffer of size 1, and then increase it. How does the behavior of the code change when the buffer is larger? (or does it?) What would you predict `num_full` to be with different buffer sizes (e.g., `-m 10`) and different numbers of produced items (e.g., `-l 100`), when you change the consumer sleep string from default (no sleep) to `-C 0,0,0,0,0,0,1`?
3. If possible, run the code on different systems (e.g., Mac OS X and Linux). Do you see different behavior across these systems?

4. Let's look at some timings of different runs. How long do you think the following execution, with one producer, three consumers, a single-entry shared buffer, and each consumer pausing at point `c3` for a second, will take?

```
prompt> ./main-one-cv-while -p 1 -c 3 -m 1 -C
        0,0,0,1,0,0,0:0,0,0,1,0,0,0:0,0,0,1,0,0,0 -l 10 -v -t
```

5. Now change the size of the shared buffer to 3 (`-m 3`). Will this make any difference in the total time?

6. Now change the location of the sleep to `c6` (this models a consumer taking something off the queue and then doing something with it for a while), again using a single-entry buffer. What time do you predict in this case?

```
prompt> ./main-one-cv-while -p 1 -c 3 -m 1 -C
        0,0,0,0,0,0,1:0,0,0,0,0,0,1:0,0,0,0,0,0,1 -l 10 -v -t
```

7. Finally, change the buffer size to 3 again (`-m 3`). What time do you predict now?

8. Now let's look at `main-one-cv-while.c`. Can you configure a sleep string, assuming a single producer, one consumer, and a buffer of size 1, to cause a problem with this code?

9. Now change the number of consumers to two. Can you construct sleep strings for the producer and the consumers so as to cause a problem in the code?

10. Now examine `main-two-cvs-if.c`. Can you cause a problem to happen in this code? Again consider the case where there is only one consumer, and then the case where there is more than one.

11. Finally, examine `main-two-cvs-while-extra-unlock.c`. What problem arises when you release the lock before doing a put or a get? Can you reliably cause such a problem to happen, given the sleep strings? What bad thing can happen?

31

Semaphores

As we know now, one needs both locks and condition variables to solve a broad range of relevant and interesting concurrency problems. One of the first people to realize this years ago was **Edsger Dijkstra** (though it is hard to know the exact history [GR92]), known among other things for his famous "shortest paths" algorithm in graph theory [D59], an early polemic on structured programming entitled "Goto Statements Considered Harmful" [D68a] (what a great title!), and, in the case we will study here, the introduction of a synchronization primitive called the **semaphore** [D68b,D72]. Indeed, Dijkstra and colleagues invented the semaphore as a single primitive for all things related to synchronization; as you will see, one can use semaphores as both locks and condition variables.

THE CRUX: HOW TO USE SEMAPHORES

How can we use semaphores instead of locks and condition variables? What is the definition of a semaphore? What is a binary semaphore? Is it straightforward to build a semaphore out of locks and condition variables? To build locks and condition variables out of semaphores?

31.1 Semaphores: A Definition

A semaphore is an object with an integer value that we can manipulate with two routines; in the POSIX standard, these routines are `sem_wait()` and `sem_post()`[1]. Because the initial value of the semaphore determines its behavior, before calling any other routine to interact with the semaphore, we must first initialize it to some value, as the code in Figure 31.1 does.

[1] Historically, `sem_wait()` was called P() by Dijkstra and `sem_post()` called V(). P() comes from "prolaag", a contraction of "probeer" (Dutch for "try") and "verlaag" ("decrease"); V() comes from the Dutch word "verhoog" which means "increase" (thanks to Mart Oskamp for this information). Sometimes, people call them down and up. Use the Dutch versions to impress your friends, or confuse them, or both.

```
1  #include <semaphore.h>
2  sem_t s;
3  sem_init(&s, 0, 1);
```

Figure 31.1: **Initializing A Semaphore**

In the figure, we declare a semaphore s and initialize it to the value 1 by passing 1 in as the third argument. The second argument to `sem_init()` will be set to 0 in all of the examples we'll see; this indicates that the semaphore is shared between threads in the same process. See the man page for details on other usages of semaphores (namely, how they can be used to synchronize access across *different* processes), which require a different value for that second argument.

After a semaphore is initialized, we can call one of two functions to interact with it, `sem_wait()` or `sem_post()`. The behavior of these two functions is seen in Figure 31.2.

For now, we are not concerned with the implementation of these routines, which clearly requires some care; with multiple threads calling into `sem_wait()` and `sem_post()`, there is the obvious need for managing these critical sections. We will now focus on how to *use* these primitives; later we may discuss how they are built.

We should discuss a few salient aspects of the interfaces here. First, we can see that `sem_wait()` will either return right away (because the value of the semaphore was one or higher when we called `sem_wait()`), or it will cause the caller to suspend execution waiting for a subsequent post. Of course, multiple calling threads may call into `sem_wait()`, and thus all be queued waiting to be woken.

Second, we can see that `sem_post()` does not wait for some particular condition to hold like `sem_wait()` does. Rather, it simply increments the value of the semaphore and then, if there is a thread waiting to be woken, wakes one of them up.

Third, the value of the semaphore, when negative, is equal to the number of waiting threads [D68b]. Though the value generally isn't seen by users of the semaphores, this invariant is worth knowing and perhaps can help you remember how a semaphore functions.

Don't worry (yet) about the seeming race conditions possible within the semaphore; assume that the actions they make are performed atomically. We will soon use locks and condition variables to do just this.

```
1  int sem_wait(sem_t *s) {
2      decrement the value of semaphore s by one
3      wait if value of semaphore s is negative
4  }
5
6  int sem_post(sem_t *s) {
7      increment the value of semaphore s by one
8      if there are one or more threads waiting, wake one
9  }
```

Figure 31.2: **Semaphore: Definitions Of Wait And Post**

```
1  sem_t m;
2  sem_init(&m, 0, X); // initialize semaphore to X; what should X be?
3
4  sem_wait(&m);
5  // critical section here
6  sem_post(&m);
```

Figure 31.3: **A Binary Semaphore (That Is, A Lock)**

31.2 Binary Semaphores (Locks)

We are now ready to use a semaphore. Our first use will be one with which we are already familiar: using a semaphore as a lock. See Figure 31.3 for a code snippet; therein, you'll see that we simply surround the critical section of interest with a `sem_wait()`/`sem_post()` pair. Critical to making this work, though, is the initial value of the semaphore `m` (initialized to `X` in the figure). What should `X` be?

... (Try thinking about it before going on) ...

Looking back at definition of the `sem_wait()` and `sem_post()` routines above, we can see that the initial value should be 1.

To make this clear, let's imagine a scenario with two threads. The first thread (Thread 0) calls `sem_wait()`; it will first decrement the value of the semaphore, changing it to 0. Then, it will wait only if the value is *not* greater than or equal to 0. Because the value is 0, `sem_wait()` will simply return and the calling thread will continue; Thread 0 is now free to enter the critical section. If no other thread tries to acquire the lock while Thread 0 is inside the critical section, when it calls `sem_post()`, it will simply restore the value of the semaphore to 1 (and not wake a waiting thread, because there are none). Figure 31.4 shows a trace of this scenario.

A more interesting case arises when Thread 0 "holds the lock" (i.e., it has called `sem_wait()` but not yet called `sem_post()`), and another thread (Thread 1) tries to enter the critical section by calling `sem_wait()`. In this case, Thread 1 will decrement the value of the semaphore to -1, and thus wait (putting itself to sleep and relinquishing the processor). When Thread 0 runs again, it will eventually call `sem_post()`, incrementing the value of the semaphore back to zero, and then wake the waiting thread (Thread 1), which will then be able to acquire the lock for itself. When Thread 1 finishes, it will again increment the value of the semaphore, restoring it to 1 again.

Value of Semaphore	Thread 0	Thread 1
1		
1	call `sem_wait()`	
0	`sem_wait()` returns	
0	`(crit sect)`	
0	call `sem_post()`	
1	`sem_post()` returns	

Figure 31.4: **Thread Trace: Single Thread Using A Semaphore**

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call sem_wait()	Running		Ready
0	sem_wait() returns	Running		Ready
0	(crit sect: begin)	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	call sem_wait()	Running
-1		Ready	decrement sem	Running
-1		Ready	(sem<0)→sleep	Sleeping
-1		Running	*Switch→T0*	Sleeping
-1	(crit sect: end)	Running		Sleeping
-1	call sem_post()	Running		Sleeping
0	increment sem	Running		Sleeping
0	wake(T1)	Running		Ready
0	sem_post() returns	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	sem_wait() returns	Running
0		Ready	(crit sect)	Running
0		Ready	call sem_post()	Running
1		Ready	sem_post() returns	Running

Figure 31.5: **Thread Trace: Two Threads Using A Semaphore**

Figure 31.5 shows a trace of this example. In addition to thread actions, the figure shows the **scheduler state** of each thread: Running, Ready (i.e., runnable but not running), and Sleeping. Note in particular that Thread 1 goes into the sleeping state when it tries to acquire the already-held lock; only when Thread 0 runs again can Thread 1 be awoken and potentially run again.

If you want to work through your own example, try a scenario where multiple threads queue up waiting for a lock. What would the value of the semaphore be during such a trace?

Thus we are able to use semaphores as locks. Because locks only have two states (held and not held), we sometimes call a semaphore used as a lock a **binary semaphore**. Note that if you are using a semaphore only in this binary fashion, it could be implemented in a simpler manner than the generalized semaphores we present here.

31.3 Semaphores As Condition Variables

Semaphores are also useful when a thread wants to halt its progress waiting for a condition to become true. For example, a thread may wish to wait for a list to become non-empty, so it can delete an element from it. In this pattern of usage, we often find a thread *waiting* for something to happen, and a different thread making that something happen and then *signaling* that it has happened, thus waking the waiting thread. Because the waiting thread (or threads) is waiting for some **condition** in the program to change, we are using the semaphore as a **condition variable**.

```
1  sem_t s;
2
3  void *
4  child(void *arg) {
5      printf("child\n");
6      sem_post(&s); // signal here: child is done
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     sem_init(&s, 0, X); // what should X be?
13     printf("parent: begin\n");
14     pthread_t c;
15     Pthread_create(c, NULL, child, NULL);
16     sem_wait(&s); // wait here for child
17     printf("parent: end\n");
18     return 0;
19 }
```

Figure 31.6: **A Parent Waiting For Its Child**

A simple example is as follows. Imagine a thread creates another thread and then wants to wait for it to complete its execution (Figure 31.6). When this program runs, we would like to see the following:

```
parent: begin
child
parent: end
```

The question, then, is how to use a semaphore to achieve this effect; as it turns out, the answer is relatively easy to understand. As you can see in the code, the parent simply calls `sem_wait()` and the child `sem_post()` to wait for the condition of the child finishing its execution to become true. However, this raises the question: what should the initial value of this semaphore be?

(Again, think about it here, instead of reading ahead)

The answer, of course, is that the value of the semaphore should be set to is 0. There are two cases to consider. First, let us assume that the parent creates the child but the child has not run yet (i.e., it is sitting in a ready queue but not running). In this case (Figure 31.7, page 6), the parent will call `sem_wait()` before the child has called `sem_post()`; we'd like the parent to wait for the child to run. The only way this will happen is if the value of the semaphore is not greater than 0; hence, 0 is the initial value. The parent runs, decrements the semaphore (to -1), then waits (sleeping). When the child finally runs, it will call `sem_post()`, increment the value of the semaphore to 0, and wake the parent, which will then return from `sem_wait()` and finish the program.

The second case (Figure 31.8, page 6) occurs when the child runs to completion before the parent gets a chance to call `sem_wait()`. In this case, the child will first call `sem_post()`, thus incrementing the value of the semaphore from 0 to 1. When the parent then gets a chance to run, it will call `sem_wait()` and find the value of the semaphore to be 1; the parent will thus decrement the value (to 0) and return from `sem_wait()` without waiting, also achieving the desired effect.

Value	Parent	State	Child	State
0	`create(Child)`	Running	*(Child exists; is runnable)*	Ready
0	call `sem_wait()`	Running		Ready
-1	`decrement sem`	Running		Ready
-1	`(sem<0)`→`sleep`	Sleeping		Ready
-1	*Switch→Child*	Sleeping	`child runs`	Running
-1		Sleeping	call `sem_post()`	Running
0		Sleeping	`increment sem`	Running
0		Ready	`wake(Parent)`	Running
0		Ready	`sem_post()` returns	Running
0		Ready	*Interrupt; Switch→Parent*	Ready
0	`sem_wait()` returns	Running		Ready

Figure 31.7: **Thread Trace: Parent Waiting For Child (Case 1)**

Value	Parent	State	Child	State
0	`create(Child)`	Running	*(Child exists; is runnable)*	Ready
0	*Interrupt; Switch→Child*	Ready	`child runs`	Running
0		Ready	call `sem_post()`	Running
1		Ready	`increment sem`	Running
1		Ready	`wake(nobody)`	Running
1		Ready	`sem_post()` returns	Running
1	`parent runs`	Running	*Interrupt; Switch→Parent*	Ready
1	call `sem_wait()`	Running		Ready
0	`decrement sem`	Running		Ready
0	`(sem≥0)`→`awake`	Running		Ready
0	`sem_wait()` returns	Running		Ready

Figure 31.8: **Thread Trace: Parent Waiting For Child (Case 2)**

31.4 The Producer/Consumer (Bounded Buffer) Problem

The next problem we will confront in this chapter is known as the **producer/consumer** problem, or sometimes as the **bounded buffer** problem [D72]. This problem is described in detail in the previous chapter on condition variables; see there for details.

First Attempt

Our first attempt at solving the problem introduces two semaphores, `empty` and `full`, which the threads will use to indicate when a buffer entry has been emptied or filled, respectively. The code for the put and get routines is in Figure 31.9, and our attempt at solving the producer and consumer problem is in Figure 31.10.

In this example, the producer first waits for a buffer to become empty in order to put data into it, and the consumer similarly waits for a buffer to become filled before using it. Let us first imagine that `MAX=1` (there is only one buffer in the array), and see if this works.

Imagine again there are two threads, a producer and a consumer. Let us examine a specific scenario on a single CPU. Assume the consumer gets to run first. Thus, the consumer will hit line c1 in the figure above, calling `sem_wait(&full)`. Because full was initialized to the value 0,

```
1  int buffer[MAX];
2  int fill = 0;
3  int use  = 0;
4
5  void put(int value) {
6      buffer[fill] = value;    // line f1
7      fill = (fill + 1) % MAX; // line f2
8  }
9
10 int get() {
11     int tmp = buffer[use];   // line g1
12     use = (use + 1) % MAX;   // line g2
13     return tmp;
14 }
```

Figure 31.9: **The Put And Get Routines**

```
1  sem_t empty;
2  sem_t full;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          sem_wait(&empty);           // line P1
8          put(i);                     // line P2
9          sem_post(&full);            // line P3
10     }
11 }
12
13 void *consumer(void *arg) {
14     int i, tmp = 0;
15     while (tmp != -1) {
16         sem_wait(&full);            // line C1
17         tmp = get();                // line C2
18         sem_post(&empty);           // line C3
19         printf("%d\n", tmp);
20     }
21 }
22
23 int main(int argc, char *argv[]) {
24     // ...
25     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
26     sem_init(&full, 0, 0);    // ... and 0 are full
27     // ...
28 }
```

Figure 31.10: **Adding The Full And Empty Conditions**

the call will decrement `full` (to -1), block the consumer, and wait for another thread to call `sem_post()` on `full`, as desired.

Assume the producer then runs. It will hit line P1, thus calling the `sem_wait(&empty)` routine. Unlike the consumer, the producer will continue through this line, because empty was initialized to the value MAX (in this case, 1). Thus, empty will be decremented to 0 and the producer will put a data value into the first entry of buffer (line P2). The producer will then continue on to P3 and call `sem_post(&full)`, changing the value of the full semaphore from -1 to 0 and waking the consumer (e.g., move it from blocked to ready).

In this case, one of two things could happen. If the producer continues to run, it will loop around and hit line P1 again. This time, however, it would block, as the empty semaphore's value is 0. If the producer instead was interrupted and the consumer began to run, it would call `sem_wait(&full)` (line c1) and find that the buffer was indeed full and thus consume it. In either case, we achieve the desired behavior.

You can try this same example with more threads (e.g., multiple producers, and multiple consumers). It should still work.

Let us now imagine that `MAX` is greater than 1 (say `MAX` = 10). For this example, let us assume that there are multiple producers and multiple consumers. We now have a problem: a race condition. Do you see where it occurs? (take some time and look for it) If you can't see it, here's a hint: look more closely at the put() and get() code.

OK, let's understand the issue. Imagine two producers (Pa and Pb) both calling into put() at roughly the same time. Assume producer Pa gets to run first, and just starts to fill the first buffer entry (fill = 0 at line f1). Before Pa gets a chance to increment the fill counter to 1, it is interrupted. Producer Pb starts to run, and at line f1 it also puts its data into the 0th element of buffer, which means that the old data there is overwritten! This is a no-no; we don't want any data from the producer to be lost.

A Solution: Adding Mutual Exclusion

As you can see, what we've forgotten here is *mutual exclusion*. The filling of a buffer and incrementing of the index into the buffer is a critical section, and thus must be guarded carefully. So let's use our friend the binary semaphore and add some locks. Figure 31.11 shows our attempt.

Now we've added some locks around the entire put()/get() parts of the code, as indicated by the `NEW LINE` comments. That seems like the right idea, but it also doesn't work. Why? Deadlock. Why does deadlock occur? Take a moment to consider it; try to find a case where deadlock arises. What sequence of steps must happen for the program to deadlock?

Avoiding Deadlock

OK, now that you figured it out, here is the answer. Imagine two threads, one producer and one consumer. The consumer gets to run first. It acquires the mutex (line c0), and then calls `sem_wait()` on the full semaphore (line c1); because there is no data yet, this call causes the consumer to block and thus yield the CPU; importantly, though, the consumer still holds the lock.

A producer then runs. It has data to produce and if it were able to run, it would be able to wake the consumer thread and all would be good. Unfortunately, the first thing it does is call `sem_wait()` on the binary mutex semaphore (line p0). The lock is already held. Hence, the producer is now stuck waiting too.

```
1  sem_t empty;
2  sem_t full;
3  sem_t mutex;
4
5  void *producer(void *arg) {
6      int i;
7      for (i = 0; i < loops; i++) {
8          sem_wait(&mutex);           // line p0 (NEW LINE)
9          sem_wait(&empty);           // line p1
10         put(i);                     // line p2
11         sem_post(&full);            // line p3
12         sem_post(&mutex);           // line p4 (NEW LINE)
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&mutex);           // line c0 (NEW LINE)
20         sem_wait(&full);            // line c1
21         int tmp = get();            // line c2
22         sem_post(&empty);           // line c3
23         sem_post(&mutex);           // line c4 (NEW LINE)
24         printf("%d\n", tmp);
25     }
26 }
27
28 int main(int argc, char *argv[]) {
29     // ...
30     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31     sem_init(&full, 0, 0);    // ... and 0 are full
32     sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock (NEW LINE)
33     // ...
34 }
```

Figure 31.11: **Adding Mutual Exclusion (Incorrectly)**

There is a simple cycle here. The consumer *holds* the mutex and is *waiting* for the someone to signal full. The producer could *signal* full but is *waiting* for the mutex. Thus, the producer and consumer are each stuck waiting for each other: a classic deadlock.

Finally, A Working Solution

To solve this problem, we simply must reduce the scope of the lock. Figure 31.12 shows the final working solution. As you can see, we simply move the mutex acquire and release to be just around the critical section; the full and empty wait and signal code is left outside. The result is a simple and working bounded buffer, a commonly-used pattern in multi-threaded programs. Understand it now; use it later. You will thank us for years to come. Or at least, you will thank us when the same question is asked on the final exam.

```
1  sem_t empty;
2  sem_t full;
3  sem_t mutex;
4
5  void *producer(void *arg) {
6      int i;
7      for (i = 0; i < loops; i++) {
8          sem_wait(&empty);           // line p1
9          sem_wait(&mutex);           // line p1.5 (MOVED MUTEX HERE...)
10         put(i);                     // line p2
11         sem_post(&mutex);           // line p2.5 (... AND HERE)
12         sem_post(&full);            // line p3
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&full);            // line c1
20         sem_wait(&mutex);           // line c1.5 (MOVED MUTEX HERE...)
21         int tmp = get();            // line c2
22         sem_post(&mutex);           // line c2.5 (... AND HERE)
23         sem_post(&empty);           // line c3
24         printf("%d\n", tmp);
25     }
26 }
27
28 int main(int argc, char *argv[]) {
29     // ...
30     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31     sem_init(&full, 0, 0);    // ... and 0 are full
32     sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
33     // ...
34 }
```

Figure 31.12: **Adding Mutual Exclusion (Correctly)**

31.5 Reader-Writer Locks

Another classic problem stems from the desire for a more flexible locking primitive that admits that different data structure accesses might require different kinds of locking. For example, imagine a number of concurrent list operations, including inserts and simple lookups. While inserts change the state of the list (and thus a traditional critical section makes sense), lookups simply *read* the data structure; as long as we can guarantee that no insert is on-going, we can allow many lookups to proceed concurrently. The special type of lock we will now develop to support this type of operation is known as a **reader-writer lock** [CHP71]. The code for such a lock is available in Figure 31.13.

The code is pretty simple. If some thread wants to update the data structure in question, it should call the new pair of synchronization operations: `rwlock_acquire_writelock()`, to acquire a write lock, and `rwlock_release_writelock()`, to release it. Internally, these simply use the `writelock` semaphore to ensure that only a single writer can ac-

```
1  typedef struct _rwlock_t {
2    sem_t lock;      // binary semaphore (basic lock)
3    sem_t writelock; // used to allow ONE writer or MANY readers
4    int   readers;   // count of readers reading in critical section
5  } rwlock_t;
6
7  void rwlock_init(rwlock_t *rw) {
8    rw->readers = 0;
9    sem_init(&rw->lock, 0, 1);
10   sem_init(&rw->writelock, 0, 1);
11 }
12
13 void rwlock_acquire_readlock(rwlock_t *rw) {
14   sem_wait(&rw->lock);
15   rw->readers++;
16   if (rw->readers == 1)
17     sem_wait(&rw->writelock); // first reader acquires writelock
18   sem_post(&rw->lock);
19 }
20
21 void rwlock_release_readlock(rwlock_t *rw) {
22   sem_wait(&rw->lock);
23   rw->readers--;
24   if (rw->readers == 0)
25     sem_post(&rw->writelock); // last reader releases writelock
26   sem_post(&rw->lock);
27 }
28
29 void rwlock_acquire_writelock(rwlock_t *rw) {
30   sem_wait(&rw->writelock);
31 }
32
33 void rwlock_release_writelock(rwlock_t *rw) {
34   sem_post(&rw->writelock);
35 }
```

Figure 31.13: **A Simple Reader-Writer Lock**

quire the lock and thus enter the critical section to update the data structure in question.

More interesting is the pair of routines to acquire and release read locks. When acquiring a read lock, the reader first acquires `lock` and then increments the `readers` variable to track how many readers are currently inside the data structure. The important step then taken within `rwlock_acquire_readlock()` occurs when the first reader acquires the lock; in that case, the reader also acquires the write lock by calling `sem_wait()` on the `writelock` semaphore, and then finally releasing the `lock` by calling `sem_post()`.

Thus, once a reader has acquired a read lock, more readers will be allowed to acquire the read lock too; however, any thread that wishes to acquire the write lock will have to wait until *all* readers are finished; the last one to exit the critical section calls `sem_post()` on "writelock" and thus enables a waiting writer to acquire the lock.

This approach works (as desired), but does have some negatives, espe-

TIP: SIMPLE AND DUMB CAN BE BETTER (HILL'S LAW)

You should never underestimate the notion that the simple and dumb approach can be the best one. With locking, sometimes a simple spin lock works best, because it is easy to implement and fast. Although something like reader/writer locks sounds cool, they are complex, and complex can mean slow. Thus, always try the simple and dumb approach first.

This idea, of appealing to simplicity, is found in many places. One early source is Mark Hill's dissertation [H87], which studied how to design caches for CPUs. Hill found that simple direct-mapped caches worked better than fancy set-associative designs (one reason is that in caching, simpler designs enable faster lookups). As Hill succinctly summarized his work: "Big and dumb is better." And thus we call this similar advice **Hill's Law**.

cially when it comes to fairness. In particular, it would be relatively easy for readers to starve writers. More sophisticated solutions to this problem exist; perhaps you can think of a better implementation? Hint: think about what you would need to do to prevent more readers from entering the lock once a writer is waiting.

Finally, it should be noted that reader-writer locks should be used with some caution. They often add more overhead (especially with more sophisticated implementations), and thus do not end up speeding up performance as compared to just using simple and fast locking primitives [CB08]. Either way, they showcase once again how we can use semaphores in an interesting and useful way.

31.6 The Dining Philosophers

One of the most famous concurrency problems posed, and solved, by Dijkstra, is known as the **dining philosopher's problem** [D71]. The problem is famous because it is fun and somewhat intellectually interesting; however, its practical utility is low. However, its fame forces its inclusion here; indeed, you might be asked about it on some interview, and you'd really hate your OS professor if you miss that question and don't get the job. Conversely, if you get the job, please feel free to send your OS professor a nice note, or some stock options.

The basic setup for the problem is this (as shown in Figure 31.14): assume there are five "philosophers" sitting around a table. Between each pair of philosophers is a single fork (and thus, five total). The philosophers each have times where they think, and don't need any forks, and times where they eat. In order to eat, a philosopher needs two forks, both the one on their left and the one on their right. The contention for these forks, and the synchronization problems that ensue, are what makes this a problem we study in concurrent programming.

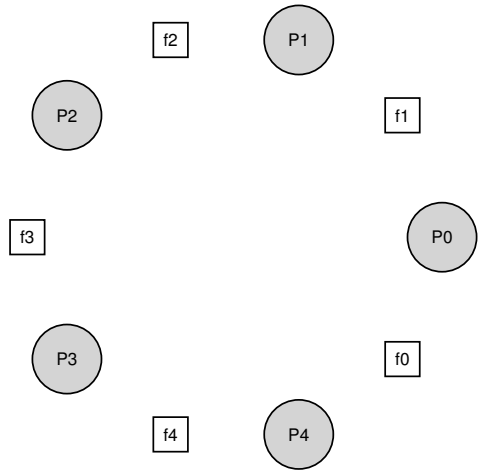


Figure 31.14: **The Dining Philosophers**

Here is the basic loop of each philosopher:

```
while (1) {
  think();
  getforks();
  eat();
  putforks();
}
```

The key challenge, then, is to write the routines `getforks()` and `putforks()` such that there is no deadlock, no philosopher starves and never gets to eat, and concurrency is high (i.e., as many philosophers can eat at the same time as possible).

Following Downey's solutions [D08], we'll use a few helper functions to get us towards a solution. They are:

```
int left(int p)  { return p; }
int right(int p) { return (p + 1) % 5; }
```

When philosopher `p` wishes to refer to the fork on their left, they simply call `left(p)`. Similarly, the fork on the right of a philosopher `p` is referred to by calling `right(p)`; the modulo operator therein handles the one case where the last philosopher (`p=4`) tries to grab the fork on their right, which is fork 0.

We'll also need some semaphores to solve this problem. Let us assume we have five, one for each fork: `sem_t forks[5]`.

```
1  void getforks() {
2    sem_wait(forks[left(p)]);
3    sem_wait(forks[right(p)]);
4  }
5
6  void putforks() {
7    sem_post(forks[left(p)]);
8    sem_post(forks[right(p)]);
9  }
```

Figure 31.15: **The `getforks()` And `putforks()` Routines**

Broken Solution

We attempt our first solution to the problem. Assume we initialize each semaphore (in the `forks` array) to a value of 1. Assume also that each philosopher knows its own number (`p`). We can thus write the `getforks()` and `putforks()` routine as shown in Figure 31.15.

The intuition behind this (broken) solution is as follows. To acquire the forks, we simply grab a "lock" on each one: first the one on the left, and then the one on the right. When we are done eating, we release them. Simple, no? Unfortunately, in this case, simple means broken. Can you see the problem that arises? Think about it.

The problem is **deadlock**. If each philosopher happens to grab the fork on their left before any philosopher can grab the fork on their right, each will be stuck holding one fork and waiting for another, forever. Specifically, philosopher 0 grabs fork 0, philosopher 1 grabs fork 1, philosopher 2 grabs fork 2, philosopher 3 grabs fork 3, and philosopher 4 grabs fork 4; all the forks are acquired, and all the philosophers are stuck waiting for a fork that another philosopher possesses. We'll study deadlock in more detail soon; for now, it is safe to say that this is not a working solution.

A Solution: Breaking The Dependency

The simplest way to attack this problem is to change how forks are acquired by at least one of the philosophers; indeed, this is how Dijkstra himself solved the problem. Specifically, let's assume that philosopher 4 (the highest numbered one) acquires the forks in a *different* order. The code to do so is as follows:

```
1  void getforks() {
2    if (p == 4) {
3      sem_wait(forks[right(p)]);
4      sem_wait(forks[left(p)]);
5    } else {
6      sem_wait(forks[left(p)]);
7      sem_wait(forks[right(p)]);
8    }
9  }
```

Because the last philosopher tries to grab right before left, there is no situation where each philosopher grabs one fork and is stuck waiting for another; the cycle of waiting is broken. Think through the ramifications of this solution, and convince yourself that it works.

```
1  typedef struct __Zem_t {
2      int value;
3      pthread_cond_t cond;
4      pthread_mutex_t lock;
5  } Zem_t;
6
7  // only one thread can call this
8  void Zem_init(Zem_t *s, int value) {
9      s->value = value;
10     Cond_init(&s->cond);
11     Mutex_init(&s->lock);
12 }
13
14 void Zem_wait(Zem_t *s) {
15     Mutex_lock(&s->lock);
16     while (s->value <= 0)
17         Cond_wait(&s->cond, &s->lock);
18     s->value--;
19     Mutex_unlock(&s->lock);
20 }
21
22 void Zem_post(Zem_t *s) {
23     Mutex_lock(&s->lock);
24     s->value++;
25     Cond_signal(&s->cond);
26     Mutex_unlock(&s->lock);
27 }
```

Figure 31.16: **Implementing Zemaphores With Locks And CVs**

There are other "famous" problems like this one, e.g., the **cigarette smoker's problem** or the **sleeping barber problem**. Most of them are just excuses to think about concurrency; some of them have fascinating names. Look them up if you are interested in learning more, or just getting more practice thinking in a concurrent manner [D08].

31.7 How To Implement Semaphores

Finally, let's use our low-level synchronization primitives, locks and condition variables, to build our own version of semaphores called ... *(drum roll here)* ... **Zemaphores**. This task is fairly straightforward, as you can see in Figure 31.16.

As you can see from the figure, we use just one lock and one condition variable, plus a state variable to track the value of the semaphore. Study the code for yourself until you really understand it. Do it!

One subtle difference between our Zemaphore and pure semaphores as defined by Dijkstra is that we don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads; indeed, the value will never be lower than zero. This behavior is easier to implement and matches the current Linux implementation.

Curiously, building locks and condition variables out of semaphores

TIP: BE CAREFUL WITH GENERALIZATION

The abstract technique of generalization can thus be quite useful in systems design, where one good idea can be made slightly broader and thus solve a larger class of problems. However, be careful when generalizing; as Lampson warns us "Don't generalize; generalizations are generally wrong" [L83].

One could view semaphores as a generalization of locks and condition variables; however, is such a generalization needed? And, given the difficulty of realizing a condition variable on top of a semaphore, perhaps this generalization is not as general as you might think.

is a much trickier proposition. Some highly experienced concurrent programmers tried to do this in the Windows environment, and many different bugs ensued [B04]. Try it yourself, and see if you can figure out why building condition variables out of semaphores is more challenging than it might appear.

31.8 Summary

Semaphores are a powerful and flexible primitive for writing concurrent programs. Some programmers use them exclusively, shunning locks and condition variables, due to their simplicity and utility.

In this chapter, we have presented just a few classic problems and solutions. If you are interested in finding out more, there are many other materials you can reference. One great (and free reference) is Allen Downey's book on concurrency and programming with semaphores [D08]. This book has lots of puzzles you can work on to improve your understanding of both semaphores in specific and concurrency in general. Becoming a real concurrency expert takes years of effort; going beyond what you learn in this class is undoubtedly the key to mastering such a topic.

References

[B04] "Implementing Condition Variables with Semaphores"
Andrew Birrell
December 2004
An interesting read on how difficult implementing CVs on top of semaphores really is, and the mistakes the author and co-workers made along the way. Particularly relevant because the group had done a ton of concurrent programming; Birrell, for example, is known for (among other things) writing various thread-programming guides.

[CB08] "Real-world Concurrency"
Bryan Cantrill and Jeff Bonwick
ACM Queue. Volume 6, No. 5. September 2008
A nice article by some kernel hackers from a company formerly known as Sun on the real problems faced in concurrent code.

[CHP71] "Concurrent Control with Readers and Writers"
P.J. Courtois, F. Heymans, D.L. Parnas
Communications of the ACM, 14:10, October 1971
The introduction of the reader-writer problem, and a simple solution. Later work introduced more complex solutions, skipped here because, well, they are pretty complex.

[D59] "A Note on Two Problems in Connexion with Graphs"
E. W. Dijkstra
Numerische Mathematik 1, 269271, 1959
Available: http://www-m3.ma.tum.de/twiki/pub/MN0506/WebHome/dijkstra.pdf
Can you believe people worked on algorithms in 1959? We can't. Even before computers were any fun to use, these people had a sense that they would transform the world...

[D68a] "Go-to Statement Considered Harmful"
E.W. Dijkstra
Communications of the ACM, volume 11(3): pages 147148, March 1968
Available: http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF
Sometimes thought as the beginning of the field of software engineering.

[D68b] "The Structure of the THE Multiprogramming System"
E.W. Dijkstra
Communications of the ACM, volume 11(5), pages 341346, 1968
One of the earliest papers to point out that systems work in computer science is an engaging intellectual endeavor. Also argues strongly for modularity in the form of layered systems.

[D72] "Information Streams Sharing a Finite Buffer"
E.W. Dijkstra
Information Processing Letters 1: 179180, 1972
Available: http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD329.PDF
Did Dijkstra invent everything? No, but maybe close. He certainly was the first to clearly write down what the problems were in concurrent code. However, it is true that practitioners in operating system design knew of many of the problems described by Dijkstra, so perhaps giving him too much credit would be a misrepresentation of history.

[D08] "The Little Book of Semaphores"
A.B. Downey
Available: http://greenteapress.com/semaphores/
A nice (and free!) book about semaphores. Lots of fun problems to solve, if you like that sort of thing.

[D71] "Hierarchical ordering of sequential processes"
E.W. Dijkstra
Available: http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF
Presents numerous concurrency problems, including the Dining Philosophers. The wikipedia page about this problem is also quite informative.

[GR92] "Transaction Processing: Concepts and Techniques"
Jim Gray and Andreas Reuter
Morgan Kaufmann, September 1992
The exact quote that we find particularly humorous is found on page 485, at the top of Section 8.8: "The first multiprocessors, circa 1960, had test and set instructions ... presumably the OS implementors worked out the appropriate algorithms, although Dijkstra is generally credited with inventing semaphores many years later."

[H87] "Aspects of Cache Memory and Instruction Buffer Performance"
Mark D. Hill
Ph.D. Dissertation, U.C. Berkeley, 1987
Hill's dissertation work, for those obsessed with caching in early systems. A great example of a quantitative dissertation.

[L83] "Hints for Computer Systems Design"
Butler Lampson
ACM Operating Systems Review, 15:5, October 1983
Lampson, a famous systems researcher, loved using hints in the design of computer systems. A hint is something that is often correct but can be wrong; in this use, a signal() is telling a waiting thread that it changed the condition that the waiter was waiting on, but not to trust that the condition will be in the desired state when the waiting thread wakes up. In this paper about hints for designing systems, one of Lampson's general hints is that you should use hints. It is not as confusing as it sounds.

32

Common Concurrency Problems

Researchers have spent a great deal of time and effort looking into concurrency bugs over many years. Much of the early work focused on **deadlock**, a topic which we've touched on in the past chapters but will now dive into deeply [C+71]. More recent work focuses on studying other types of common concurrency bugs (i.e., non-deadlock bugs). In this chapter, we take a brief look at some example concurrency problems found in real code bases, to better understand what problems to look out for. And thus our central issue for this chapter:

CRUX: HOW TO HANDLE COMMON CONCURRENCY BUGS

Concurrency bugs tend to come in a variety of common patterns. Knowing which ones to look out for is the first step to writing more robust, correct concurrent code.

32.1 What Types Of Bugs Exist?

The first, and most obvious, question is this: what types of concurrency bugs manifest in complex, concurrent programs? This question is difficult to answer in general, but fortunately, some others have done the work for us. Specifically, we rely upon a study by Lu et al. [L+08], which analyzes a number of popular concurrent applications in great detail to understand what types of bugs arise in practice.

The study focuses on four major and important open-source applications: MySQL (a popular database management system), Apache (a well-known web server), Mozilla (the famous web browser), and OpenOffice (a free version of the MS Office suite, which some people actually use). In the study, the authors examine concurrency bugs that have been found and fixed in each of these code bases, turning the developers' work into a quantitative bug analysis; understanding these results can help you understand what types of problems actually occur in mature code bases.

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
OpenOffice	Office Suite	6	2
Total		74	31

Figure 32.1: **Bugs In Modern Applications**

Figure 32.1 shows a summary of the bugs Lu and colleagues studied. From the figure, you can see that there were 105 total bugs, most of which were not deadlock (74); the remaining 31 were deadlock bugs. Further, you can see that the number of bugs studied from each application; while OpenOffice only had 8 total concurrency bugs, Mozilla had nearly 60.

We now dive into these different classes of bugs (non-deadlock, deadlock) a bit more deeply. For the first class of non-deadlock bugs, we use examples from the study to drive our discussion. For the second class of deadlock bugs, we discuss the long line of work that has been done in either preventing, avoiding, or handling deadlock.

32.2 Non-Deadlock Bugs

Non-deadlock bugs make up a majority of concurrency bugs, according to Lu's study. But what types of bugs are these? How do they arise? How can we fix them? We now discuss the two major types of non-deadlock bugs found by Lu et al.: **atomicity violation** bugs and **order violation** bugs.

Atomicity-Violation Bugs

The first type of problem encountered is referred to as an **atomicity violation**. Here is a simple example, found in MySQL. Before reading the explanation, try figuring out what the bug is. Do it!

```
1  Thread 1::
2  if (thd->proc_info) {
3    ...
4    fputs(thd->proc_info, ...);
5    ...
6  }
7
8  Thread 2::
9  thd->proc_info = NULL;
```

In the example, two different threads access the field `proc_info` in the structure `thd`. The first thread checks if the value is non-NULL and then prints its value; the second thread sets it to NULL. Clearly, if the first thread performs the check but then is interrupted before the call to `fputs`, the second thread could run in-between, thus setting the pointer to NULL; when the first thread resumes, it will crash, as a NULL pointer will be dereferenced by `fputs`.

The more formal definition of an atomicity violation, according to Lu et al, is this: "The desired serializability among multiple memory accesses is violated (i.e. a code region is intended to be atomic, but the atomicity is not enforced during execution)." In our example above, the code has an *atomicity assumption* (in Lu's words) about the check for non-NULL of `proc_info` and the usage of `proc_info` in the `fputs()` call; when the assumption is incorrect, the code will not work as desired.

Finding a fix for this type of problem is often (but not always) straightforward. Can you think of how to fix the code above?

In this solution, we simply add locks around the shared-variable references, ensuring that when either thread accesses the `proc_info` field, it has a lock held (`proc_info_lock`). Of course, any other code that accesses the structure should also acquire this lock before doing so.

```
1  pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;
2
3  Thread 1::
4  pthread_mutex_lock(&proc_info_lock);
5  if (thd->proc_info) {
6    ...
7    fputs(thd->proc_info, ...);
8    ...
9  }
10 pthread_mutex_unlock(&proc_info_lock);
11
12 Thread 2::
13 pthread_mutex_lock(&proc_info_lock);
14 thd->proc_info = NULL;
15 pthread_mutex_unlock(&proc_info_lock);
```

Order-Violation Bugs

Another common type of non-deadlock bug found by Lu et al. is known as an **order violation**. Here is another simple example; once again, see if you can figure out why the code below has a bug in it.

```
1  Thread 1::
2  void init() {
3     ...
4     mThread = PR_CreateThread(mMain, ...);
5     ...
6  }
7
8  Thread 2::
9  void mMain(...) {
10     ...
11     mState = mThread->State;
12     ...
13 }
```

As you probably figured out, the code in Thread 2 seems to assume that the variable `mThread` has already been initialized (and is not NULL); however, if Thread 2 runs immediately once created, the value of `mThread` will not be set when it is accessed within `mMain()` in Thread 2, and will

likely crash with a NULL-pointer dereference. Note that we assume the value of `mThread` is initially NULL; if not, even stranger things could happen as arbitrary memory locations are accessed through the dereference in Thread 2.

The more formal definition of an order violation is this: "The desired order between two (groups of) memory accesses is flipped (i.e., A should always be executed before B, but the order is not enforced during execution)" [L+08].

The fix to this type of bug is generally to enforce ordering. As we discussed in detail previously, using **condition variables** is an easy and robust way to add this style of synchronization into modern code bases. In the example above, we could thus rewrite the code as follows:

```
1  pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t  mtCond = PTHREAD_COND_INITIALIZER;
3  int mtInit             = 0;
4
5  Thread 1::
6  void init() {
7      ...
8      mThread = PR_CreateThread(mMain, ...);
9
10     // signal that the thread has been created...
11     pthread_mutex_lock(&mtLock);
12     mtInit = 1;
13     pthread_cond_signal(&mtCond);
14     pthread_mutex_unlock(&mtLock);
15     ...
16 }
17
18 Thread 2::
19 void mMain(...) {
20      ...
21      // wait for the thread to be initialized...
22      pthread_mutex_lock(&mtLock);
23      while (mtInit == 0)
24          pthread_cond_wait(&mtCond, &mtLock);
25      pthread_mutex_unlock(&mtLock);
26
27      mState = mThread->State;
28      ...
29 }
```

In this fixed-up code sequence, we have added a lock (`mtLock`) and corresponding condition variable (`mtCond`), as well as a state variable (`mtInit`). When the initialization code runs, it sets the state of `mtInit` to 1 and signals that it has done so. If Thread 2 had run before this point, it will be waiting for this signal and corresponding state change; if it runs later, it will check the state and see that the initialization has already occurred (i.e., `mtInit` is set to 1), and thus continue as is proper. Note that we could likely use `mThread` as the state variable itself, but do not do so for the sake of simplicity here. When ordering matters between threads, condition variables (or semaphores) can come to the rescue.

Non-Deadlock Bugs: Summary

A large fraction (97%) of non-deadlock bugs studied by Lu et al. are either atomicity or order violations. Thus, by carefully thinking about these types of bug patterns, programmers can likely do a better job of avoiding them. Moreover, as more automated code-checking tools develop, they should likely focus on these two types of bugs as they constitute such a large fraction of non-deadlock bugs found in deployment.

Unfortunately, not all bugs are as easily fixable as the examples we looked at above. Some require a deeper understanding of what the program is doing, or a larger amount of code or data structure reorganization to fix. Read Lu et al.'s excellent (and readable) paper for more details.

32.3 Deadlock Bugs

Beyond the concurrency bugs mentioned above, a classic problem that arises in many concurrent systems with complex locking protocols is known as **deadlock**. Deadlock occurs, for example, when a thread (say Thread 1) is holding a lock (`L1`) and waiting for another one (`L2`); unfortunately, the thread (Thread 2) that holds lock `L2` is waiting for `L1` to be released. Here is a code snippet that demonstrates such a potential deadlock:

```
Thread 1:                    Thread 2:
pthread_mutex_lock(L1);      pthread_mutex_lock(L2);
pthread_mutex_lock(L2);      pthread_mutex_lock(L1);
```

Note that if this code runs, deadlock does not necessarily occur; rather, it may occur, if, for example, Thread 1 grabs lock `L1` and then a context switch occurs to Thread 2. At that point, Thread 2 grabs `L2`, and tries to acquire `L1`. Thus we have a deadlock, as each thread is waiting for the other and neither can run. See Figure 32.2 for a graphical depiction; the presence of a **cycle** in the graph is indicative of the deadlock.

The figure should make clear the problem. How should programmers write code so as to handle deadlock in some way?

CRUX: HOW TO DEAL WITH DEADLOCK

How should we build systems to prevent, avoid, or at least detect and recover from deadlock? Is this a real problem in systems today?

Why Do Deadlocks Occur?

As you may be thinking, simple deadlocks such as the one above seem readily avoidable. For example, if Thread 1 and 2 both made sure to grab locks in the same order, the deadlock would never arise. So why do deadlocks happen?

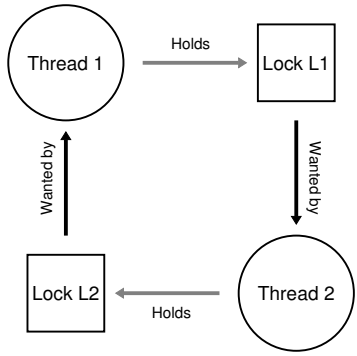


Figure 32.2: **The Deadlock Dependency Graph**

One reason is that in large code bases, complex dependencies arise between components. Take the operating system, for example. The virtual memory system might need to access the file system in order to page in a block from disk; the file system might subsequently require a page of memory to read the block into and thus contact the virtual memory system. Thus, the design of locking strategies in large systems must be carefully done to avoid deadlock in the case of circular dependencies that may occur naturally in the code.

Another reason is due to the nature of **encapsulation**. As software developers, we are taught to hide details of implementations and thus make software easier to build in a modular way. Unfortunately, such modularity does not mesh well with locking. As Jula et al. point out [J+08], some seemingly innocuous interfaces almost invite you to deadlock. For example, take the Java Vector class and the method `AddAll()`. This routine would be called as follows:

```
Vector v1, v2;
v1.AddAll(v2);
```

Internally, because the method needs to be multi-thread safe, locks for both the vector being added to (v1) and the parameter (v2) need to be acquired. The routine acquires said locks in some arbitrary order (say v1 then v2) in order to add the contents of v2 to v1. If some other thread calls `v2.AddAll(v1)` at nearly the same time, we have the potential for deadlock, all in a way that is quite hidden from the calling application.

Conditions for Deadlock

Four conditions need to hold for a deadlock to occur [C+71]:

- **Mutual exclusion:** Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).
- **Hold-and-wait:** Threads hold resources allocated to them (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire).
- **No preemption:** Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.
- **Circular wait:** There exists a circular chain of threads such that each thread holds one or more resources (e.g., locks) that are being requested by the next thread in the chain.

If any of these four conditions are not met, deadlock cannot occur. Thus, we first explore techniques to *prevent* deadlock; each of these strategies seeks to prevent one of the above conditions from arising and thus is one approach to handling the deadlock problem.

Prevention

Circular Wait

Probably the most practical prevention technique (and certainly one that is frequently employed) is to write your locking code such that you never induce a circular wait. The most straightforward way to do that is to provide a **total ordering** on lock acquisition. For example, if there are only two locks in the system (`L1` and `L2`), you can prevent deadlock by always acquiring `L1` before `L2`. Such strict ordering ensures that no cyclical wait arises; hence, no deadlock.

Of course, in more complex systems, more than two locks will exist, and thus total lock ordering may be difficult to achieve (and perhaps is unnecessary anyhow). Thus, a **partial ordering** can be a useful way to structure lock acquisition so as to avoid deadlock. An excellent real example of partial lock ordering can be seen in the memory mapping code in Linux [T+94]; the comment at the top of the source code reveals ten different groups of lock acquisition orders, including simple ones such as "`i_mutex` before `i_mmap_mutex`" and more complex orders such as "`i_mmap_mutex` before `private_lock` before `swap_lock` before `mapping->tree_lock`".

As you can imagine, both total and partial ordering require careful design of locking strategies and must be constructed with great care. Further, ordering is just a convention, and a sloppy programmer can easily ignore the locking protocol and potentially cause deadlock. Finally, lock

TIP: ENFORCE LOCK ORDERING BY LOCK ADDRESS

In some cases, a function must grab two (or more) locks; thus, we know we must be careful or deadlock could arise. Imagine a function that is called as follows: `do_something(mutex_t *m1, mutex_t *m2)`. If the code always grabs `m1` before `m2` (or always `m2` before `m1`), it could deadlock, because one thread could call `do_something(L1, L2)` while another thread could call `do_something(L2, L1)`.

To avoid this particular issue, the clever programmer can use the *address* of each lock as a way of ordering lock acquisition. By acquiring locks in either high-to-low or low-to-high address order, `do_something()` can guarantee that it always acquires locks in the same order, regardless of which order they are passed in. The code would look something like this:

```
if (m1 > m2) {  // grab locks in high-to-low address order
  pthread_mutex_lock(m1);
  pthread_mutex_lock(m2);
} else {
  pthread_mutex_lock(m2);
  pthread_mutex_lock(m1);
}
// Code assumes that m1 != m2 (it is not the same lock)
```

By using this simple technique, a programmer can ensure a simple and efficient deadlock-free implementation of multi-lock acquisition.

ordering requires a deep understanding of the code base, and how various routines are called; just one mistake could result in the "D" word[1].

Hold-and-wait

The hold-and-wait requirement for deadlock can be avoided by acquiring all locks at once, atomically. In practice, this could be achieved as follows:

```
1  pthread_mutex_lock(prevention);   // begin lock acquistion
2  pthread_mutex_lock(L1);
3  pthread_mutex_lock(L2);
4  ...
5  pthread_mutex_unlock(prevention); // end
```

By first grabbing the lock `prevention`, this code guarantees that no untimely thread switch can occur in the midst of lock acquisition and thus deadlock can once again be avoided. Of course, it requires that any time any thread grabs a lock, it first acquires the global prevention lock. For example, if another thread was trying to grab locks `L1` and `L2` in a different order, it would be OK, because it would be holding the prevention lock while doing so.

[1] Hint: "D" stands for "Deadlock".

Note that the solution is problematic for a number of reasons. As before, encapsulation works against us: when calling a routine, this approach requires us to know exactly which locks must be held and to acquire them ahead of time. This technique also is likely to decrease concurrency as all locks must be acquired early on (at once) instead of when they are truly needed.

No Preemption

Because we generally view locks as held until unlock is called, multiple lock acquisition often gets us into trouble because when waiting for one lock we are holding another. Many thread libraries provide a more flexible set of interfaces to help avoid this situation. Specifically, the routine `pthread_mutex_trylock()` either grabs the lock (if it is available) and returns success or returns an error code indicating the lock is held; in the latter case, you can try again later if you want to grab that lock.

Such an interface could be used as follows to build a deadlock-free, ordering-robust lock acquisition protocol:

```
1 top:
2   pthread_mutex_lock(L1);
3   if (pthread_mutex_trylock(L2) != 0) {
4     pthread_mutex_unlock(L1);
5     goto top;
6   }
```

Note that another thread could follow the same protocol but grab the locks in the other order (`L2` then `L1`) and the program would still be deadlock free. One new problem does arise, however: **livelock**. It is possible (though perhaps unlikely) that two threads could both be repeatedly attempting this sequence and repeatedly failing to acquire both locks. In this case, both systems are running through this code sequence over and over again (and thus it is not a deadlock), but progress is not being made, hence the name livelock. There are solutions to the livelock problem, too: for example, one could add a random delay before looping back and trying the entire thing over again, thus decreasing the odds of repeated interference among competing threads.

One final point about this solution: it skirts around the hard parts of using a trylock approach. The first problem that would likely exist again arises due to encapsulation: if one of these locks is buried in some routine that is getting called, the jump back to the beginning becomes more complex to implement. If the code had acquired some resources (other than `L1`) along the way, it must make sure to carefully release them as well; for example, if after acquiring `L1`, the code had allocated some memory, it would have to release that memory upon failure to acquire `L2`, before jumping back to the top to try the entire sequence again. However, in limited circumstances (e.g., the Java vector method mentioned earlier), this type of approach could work well.

Mutual Exclusion

The final prevention technique would be to avoid the need for mutual exclusion at all. In general, we know this is difficult, because the code we wish to run does indeed have critical sections. So what can we do?

Herlihy had the idea that one could design various data structures without locks at all [H91, H93]. The idea behind these **lock-free** (and related **wait-free**) approaches here is simple: using powerful hardware instructions, you can build data structures in a manner that does not require explicit locking.

As a simple example, let us assume we have a compare-and-swap instruction, which as you may recall is an atomic instruction provided by the hardware that does the following:

```
1  int CompareAndSwap(int *address, int expected, int new) {
2    if (*address == expected) {
3      *address = new;
4      return 1; // success
5    }
6    return 0; // failure
7  }
```

Imagine we now wanted to atomically increment a value by a certain amount. We could do it as follows:

```
1  void AtomicIncrement(int *value, int amount) {
2    do {
3      int old = *value;
4    } while (CompareAndSwap(value, old, old + amount) == 0);
5  }
```

Instead of acquiring a lock, doing the update, and then releasing it, we have instead built an approach that repeatedly tries to update the value to the new amount and uses the compare-and-swap to do so. In this manner, no lock is acquired, and no deadlock can arise (though livelock is still a possibility).

Let us consider a slightly more complex example: list insertion. Here is code that inserts at the head of a list:

```
1  void insert(int value) {
2    node_t *n = malloc(sizeof(node_t));
3    assert(n != NULL);
4    n->value = value;
5    n->next  = head;
6    head     = n;
7  }
```

This code performs a simple insertion, but if called by multiple threads at the "same time", has a race condition (see if you can figure out why). Of course, we could solve this by surrounding this code with a lock acquire and release:

```
1  void insert(int value) {
2    node_t *n = malloc(sizeof(node_t));
3    assert(n != NULL);
4    n->value = value;
5    pthread_mutex_lock(listlock);   // begin critical section
6    n->next  = head;
7    head     = n;
8    pthread_mutex_unlock(listlock); // end critical section
9  }
```

In this solution, we are using locks in the traditional manner[2]. Instead, let us try to perform this insertion in a lock-free manner simply using the compare-and-swap instruction. Here is one possible approach:

```
1  void insert(int value) {
2    node_t *n = malloc(sizeof(node_t));
3    assert(n != NULL);
4    n->value = value;
5    do {
6      n->next = head;
7    } while (CompareAndSwap(&head, n->next, n) == 0);
8  }
```

The code here updates the next pointer to point to the current head, and then tries to swap the newly-created node into position as the new head of the list. However, this will fail if some other thread successfully swapped in a new head in the meanwhile, causing this thread to retry again with the new head.

Of course, building a useful list requires more than just a list insert, and not surprisingly building a list that you can insert into, delete from, and perform lookups on in a lock-free manner is non-trivial. Read the rich literature on lock-free and wait-free synchronization to learn more [H01, H91, H93].

Deadlock Avoidance via Scheduling

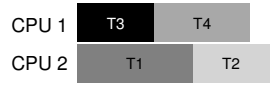
Instead of deadlock prevention, in some scenarios deadlock **avoidance** is preferable. Avoidance requires some global knowledge of which locks various threads might grab during their execution, and subsequently schedules said threads in a way as to guarantee no deadlock can occur.

For example, assume we have two processors and four threads which must be scheduled upon them. Assume further we know that Thread 1 (T1) grabs locks `L1` and `L2` (in some order, at some point during its execution), T2 grabs `L1` and `L2` as well, T3 grabs just `L2`, and T4 grabs no locks at all. We can show these lock acquisition demands of the threads in tabular form:

[2]The astute reader might be asking why we grabbed the lock so late, instead of right when entering `insert()`; can you, astute reader, figure out why that is likely correct? What assumptions does the code make, for example, about the call to `malloc()`?

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

A smart scheduler could thus compute that as long as T1 and T2 are not run at the same time, no deadlock could ever arise. Here is one such schedule:



Note that it is OK for (T3 and T1) or (T3 and T2) to overlap. Even though T3 grabs lock `L2`, it can never cause a deadlock by running concurrently with other threads because it only grabs one lock.

Let's look at one more example. In this one, there is more contention for the same resources (again, locks `L1` and `L2`), as indicated by the following contention table:

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

In particular, threads T1, T2, and T3 all need to grab both locks `L1` and `L2` at some point during their execution. Here is a possible schedule that guarantees that no deadlock could ever occur:

As you can see, static scheduling leads to a conservative approach where T1, T2, and T3 are all run on the same processor, and thus the total time to complete the jobs is lengthened considerably. Though it may have been possible to run these tasks concurrently, the fear of deadlock prevents us from doing so, and the cost is performance.

One famous example of an approach like this is Dijkstra's Banker's Algorithm [D64], and many similar approaches have been described in the literature. Unfortunately, they are only useful in very limited environments, for example, in an embedded system where one has full knowledge of the entire set of tasks that must be run and the locks that they need. Further, such approaches can limit concurrency, as we saw in the second example above. Thus, avoidance of deadlock via scheduling is not a widely-used general-purpose solution.

Detect and Recover

One final general strategy is to allow deadlocks to occasionally occur, and then take some action once such a deadlock has been detected. For example, if an OS froze once a year, you would just reboot it and get happily (or

TIP: DON'T ALWAYS DO IT PERFECTLY (TOM WEST'S LAW)
Tom West, famous as the subject of the classic computer-industry book *Soul of a New Machine* [K81], says famously: "Not everything worth doing is worth doing well", which is a terrific engineering maxim. If a bad thing happens rarely, certainly one should not spend a great deal of effort to prevent it, particularly if the cost of the bad thing occurring is small. If, on the other hand, you are building a space shuttle, and the cost of something going wrong is the space shuttle blowing up, well, perhaps you should ignore this piece of advice.

grumpily) on with your work. If deadlocks are rare, such a non-solution is indeed quite pragmatic.

Many database systems employ deadlock detection and recovery techniques. A deadlock detector runs periodically, building a resource graph and checking it for cycles. In the event of a cycle (deadlock), the system needs to be restarted. If more intricate repair of data structures is first required, a human being may be involved to ease the process.

More detail on database concurrency, deadlock, and related issues can be found elsewhere [B+87, K87]. Read these works, or better yet, take a course on databases to learn more about this rich and interesting topic.

32.4 Summary

In this chapter, we have studied the types of bugs that occur in concurrent programs. The first type, non-deadlock bugs, are surprisingly common, but often are easier to fix. They include atomicity violations, in which a sequence of instructions that should have been executed together was not, and order violations, in which the needed order between two threads was not enforced.

We have also briefly discussed deadlock: why it occurs, and what can be done about it. The problem is as old as concurrency itself, and many hundreds of papers have been written about the topic. The best solution in practice is to be careful, develop a lock acquisition order, and thus prevent deadlock from occurring in the first place. Wait-free approaches also have promise, as some wait-free data structures are now finding their way into commonly-used libraries and critical systems, including Linux. However, their lack of generality and the complexity to develop a new wait-free data structure will likely limit the overall utility of this approach. Perhaps the best solution is to develop new concurrent programming models: in systems such as MapReduce (from Google) [GD02], programmers can describe certain types of parallel computations without any locks whatsoever. Locks are problematic by their very nature; perhaps we should seek to avoid using them unless we truly must.

References

[B+87] "Concurrency Control and Recovery in Database Systems"
Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman
Addison-Wesley, 1987
The classic text on concurrency in database management systems. As you can tell, understanding concurrency, deadlock, and other topics in the world of databases is a world unto itself. Study it and find out for yourself.

[C+71] "System Deadlocks"
E.G. Coffman, M.J. Elphick, A. Shoshani
ACM Computing Surveys, 3:2, June 1971
The classic paper outlining the conditions for deadlock and how you might go about dealing with it. There are certainly some earlier papers on this topic; see the references within this paper for details.

[D64] "Een algorithme ter voorkoming van de dodelijke omarming"
Edsger Dijkstra
Circulated privately, around 1964
Available: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF
Indeed, not only did Dijkstra come up with a number of solutions to the deadlock problem, he was the first to note its existence, at least in written form. However, he called it the "deadly embrace", which (thankfully) did not catch on.

[GD02] "MapReduce: Simplified Data Processing on Large Clusters"
Sanjay Ghemawhat and Jeff Dean
OSDI '04, San Francisco, CA, October 2004
The MapReduce paper ushered in the era of large-scale data processing, and proposes a framework for performing such computations on clusters of generally unreliable machines.

[H01] "A Pragmatic Implementation of Non-blocking Linked-lists"
Tim Harris
International Conference on Distributed Computing (DISC), 2001
A relatively modern example of the difficulties of building something as simple as a concurrent linked list without locks.

[H91] "Wait-free Synchronization"
Maurice Herlihy
ACM TOPLAS, 13:1, January 1991
Herlihy's work pioneers the ideas behind wait-free approaches to writing concurrent programs. These approaches tend to be complex and hard, often more difficult than using locks correctly, probably limiting their success in the real world.

[H93] "A Methodology for Implementing Highly Concurrent Data Objects"
Maurice Herlihy
ACM TOPLAS, 15:5, November 1993
A nice overview of lock-free and wait-free structures. Both approaches eschew locks, but wait-free approaches are harder to realize, as they try to ensure than any operation on a concurrent structure will terminate in a finite number of steps (e.g., no unbounded looping).

[J+08] "Deadlock Immunity: Enabling Systems To Defend Against Deadlocks"
Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, George Candea
OSDI '08, San Diego, CA, December 2008
An excellent recent paper on deadlocks and how to avoid getting caught in the same ones over and over again in a particular system.

[K81] "Soul of a New Machine"
Tracy Kidder, 1980
A must-read for any systems builder or engineer, detailing the early days of how a team inside Data General (DG), led by Tom West, worked to produce a "new machine." Kidder's other books are also excellent, including Mountains beyond Mountains*. Or maybe you don't agree with us, comma?*

[K87] "Deadlock Detection in Distributed Databases"
Edgar Knapp
ACM Computing Surveys, 19:4, December 1987
An excellent overview of deadlock detection in distributed database systems. Also points to a number of other related works, and thus is a good place to start your reading.

[L+08] "Learning from Mistakes — A Comprehensive Study on
Real World Concurrency Bug Characteristics"
Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou
ASPLOS '08, March 2008, Seattle, Washington
The first in-depth study of concurrency bugs in real software, and the basis for this chapter. Look at Y.Y. Zhou's or Shan Lu's web pages for many more interesting papers on bugs.

[T+94] "Linux File Memory Map Code"
Linus Torvalds and many others
Available: http://lxr.free-electrons.com/source/mm/filemap.c
Thanks to Michael Walfish (NYU) for pointing out this precious example. The real world, as you can see in this file, can be a bit more complex than the simple clarity found in textbooks...

Homework

This homework lets you explore some real code that deadlocks (or avoids deadlock). The different versions of code correspond to different approaches to avoiding deadlock in a simplified `vector_add()` routine. Specifically:

- `vector-deadlock.c`: This version of `vector_add()` does not try to avoid deadlock and thus may indeed do so.
- `vector-global-order.c`: This version acquires locks in a global order to avoid deadlock.
- `vector-try-wait.c`: This version is willing to release a lock when it senses deadlock might occur.
- `vector-avoid-hold-and-wait.c`: This version uses a global lock around lock acquisition to avoid deadlock.
- `vector-nolock.c`: This version uses an atomic fetch-and-add instead of locks.

See the README for details on these programs and their common substrate.

Questions

1. First let's make sure you understand how the programs generally work, and some of the key options. Study the code in the file called `vector-deadlock.c`, as well as in `main-common.c` and related files.

   Now, run `./vector-deadlock -n 2 -l 1 -v`, which instantiates two threads (`-n 2`), each of which does one vector add (`-l 1`), and does so in verbose mode (`-v`). Make sure you understand the output. How does the output change from run to run?

2. Now add the `-d` flag, and change the number of loops (`-l`) from 1 to higher numbers. What happens? Does the code (always) deadlock?

3. How does changing the number of threads (`-n`) change the outcome of the program? Are there any values of `-n` that ensure no deadlock occurs?

4. Now examine the code in `vector-global-order.c`. First, make sure you understand what the code is trying to do; do you understand why the code avoids deadlock? Also, why is there a special case in this `vector_add()` routine when the source and destination vectors are the same?

5. Now run the code with the following flags: `-t -n 2 -l 100000 -d`. How long does the code take to complete? How does the total time change when you increase the number of loops, or the number of threads?

6. What happens if you turn on the parallelism flag (`-p`)? How much would you expect performance to change when each thread is working on adding different vectors (which is what `-p` enables) versus working on the same ones?

7. Now let's study `vector-try-wait.c`. First make sure you understand the code. Is the first call to `pthread_mutex_trylock()` really needed?

   Now run the code. How fast does it run compared to the global order approach? How does the number of retries, as counted by the code, change as the number of threads increases?

8. Now let's look at `vector-avoid-hold-and-wait.c`. What is the main problem with this approach? How does its performance compare to the other versions, when running both with `-p` and without it?

9. Finally, let's look at `vector-nolock.c`. This version doesn't use locks at all; does it provide the exact same semantics as the other versions? Why or why not?

10. Now compare its performance to the other versions, both when threads are working on the same two vectors (no `-p`) and when each thread is working on separate vectors (`-p`). How does this no-lock version perform?

33

Event-based Concurrency (Advanced)

Thus far, we've written about concurrency as if the only way to build concurrent applications is to use threads. Like many things in life, this is not completely true. Specifically, a different style of concurrent programming is often used in both GUI-based applications [O96] as well as some types of internet servers [PDZ99]. This style, known as **event-based concurrency**, has become popular in some modern systems, including server-side frameworks such as **node.js** [N13], but its roots are found in C/UNIX systems that we'll discuss below.

The problem that event-based concurrency addresses is two-fold. The first is that managing concurrency correctly in multi-threaded applications can be challenging; as we've discussed, missing locks, deadlock, and other nasty problems can arise. The second is that in a multi-threaded application, the developer has little or no control over what is scheduled at a given moment in time; rather, the programmer simply creates threads and then hopes that the underlying OS schedules them in a reasonable manner across available CPUs. Given the difficulty of building a general-purpose scheduler that works well in all cases for all workloads, sometimes the OS will schedule work in a manner that is less than optimal. The crux:

THE CRUX:
HOW TO BUILD CONCURRENT SERVERS WITHOUT THREADS
How can we build a concurrent server without using threads, and thus retain control over concurrency as well as avoid some of the problems that seem to plague multi-threaded applications?

33.1 The Basic Idea: An Event Loop

The basic approach we'll use, as stated above, is called **event-based concurrency**. The approach is quite simple: you simply wait for something (i.e., an "event") to occur; when it does, you check what type of

event it is and do the small amount of work it requires (which may include issuing I/O requests, or scheduling other events for future handling, etc.). That's it!

Before getting into the details, let's first examine what a canonical event-based server looks like. Such applications are based around a simple construct known as the **event loop**. Pseudocode for an event loop looks like this:

```
while (1) {
    events = getEvents();
    for (e in events)
        processEvent(e);
}
```

It's really that simple. The main loop simply waits for something to do (by calling `getEvents()` in the code above) and then, for each event returned, processes them, one at a time; the code that processes each event is known as an **event handler**. Importantly, when a handler processes an event, it is the only activity taking place in the system; thus, deciding which event to handle next is equivalent to scheduling. This explicit control over scheduling is one of the fundamental advantages of the event-based approach.

But this discussion leaves us with a bigger question: how exactly does an event-based server determine which events are taking place, in particular with regards to network and disk I/O? Specifically, how can an event server tell if a message has arrived for it?

33.2 An Important API: `select()` (or `poll()`)

With that basic event loop in mind, we next must address the question of how to receive events. In most systems, a basic API is available, via either the `select()` or `poll()` system calls.

What these interfaces enable a program to do is simple: check whether there is any incoming I/O that should be attended to. For example, imagine that a network application (such as a web server) wishes to check whether any network packets have arrived, in order to service them. These system calls let you do exactly that.

Take `select()` for example. The manual page (on Mac OS X) describes the API in this manner:

```
int select(int nfds,
           fd_set *restrict readfds,
           fd_set *restrict writefds,
           fd_set *restrict errorfds,
           struct timeval *restrict timeout);
```

The actual description from the man page: *select() examines the I/O descriptor sets whose addresses are passed in readfds, writefds, and errorfds to see if some of their descriptors are ready for reading, are ready for writing, or have*

ASIDE: **BLOCKING VS. NON-BLOCKING INTERFACES**
Blocking (or **synchronous**) interfaces do all of their work before returning to the caller; non-blocking (or **asynchronous**) interfaces begin some work but return immediately, thus letting whatever work that needs to be done get done in the background.

The usual culprit in blocking calls is I/O of some kind. For example, if a call must read from disk in order to complete, it might block, waiting for the I/O request that has been sent to the disk to return.

Non-blocking interfaces can be used in any style of programming (e.g., with threads), but are essential in the event-based approach, as a call that blocks will halt all progress.

an exceptional condition pending, respectively. The first nfds descriptors are checked in each set, i.e., the descriptors from 0 through nfds-1 in the descriptor sets are examined. On return, select() replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. select() returns the total number of ready descriptors in all the sets.

A couple of points about `select()`. First, note that it lets you check whether descriptors can be *read* from as well as *written* to; the former lets a server determine that a new packet has arrived and is in need of processing, whereas the latter lets the service know when it is OK to reply (i.e., the outbound queue is not full).

Second, note the timeout argument. One common usage here is to set the timeout to NULL, which causes `select()` to block indefinitely, until some descriptor is ready. However, more robust servers will usually specify some kind of timeout; one common technique is to set the timeout to zero, and thus use the call to `select()` to return immediately.

The `poll()` system call is quite similar. See its manual page, or Stevens and Rago [SR05], for details.

Either way, these basic primitives give us a way to build a non-blocking event loop, which simply checks for incoming packets, reads from sockets with messages upon them, and replies as needed.

33.3 Using `select()`

To make this more concrete, let's examine how to use `select()` to see which network descriptors have incoming messages upon them. Figure 33.1 shows a simple example.

This code is actually fairly simple to understand. After some initialization, the server enters an infinite loop. Inside the loop, it uses the `FD_ZERO()` macro to first clear the set of file descriptors, and then uses `FD_SET()` to include all of the file descriptors from `minFD` to `maxFD` in the set. This set of descriptors might represent, for example, all of the net-

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/time.h>
4  #include <sys/types.h>
5  #include <unistd.h>
6
7  int main(void) {
8      // open and set up a bunch of sockets (not shown)
9      // main loop
10     while (1) {
11         // initialize the fd_set to all zero
12         fd_set readFDs;
13         FD_ZERO(&readFDs);
14
15         // now set the bits for the descriptors
16         // this server is interested in
17         // (for simplicity, all of them from min to max)
18         int fd;
19         for (fd = minFD; fd < maxFD; fd++)
20             FD_SET(fd, &readFDs);
21
22         // do the select
23         int rc = select(maxFD+1, &readFDs, NULL, NULL, NULL);
24
25         // check which actually have data using FD_ISSET()
26         int fd;
27         for (fd = minFD; fd < maxFD; fd++)
28             if (FD_ISSET(fd, &readFDs))
29                 processFD(fd);
30     }
31 }
```

Figure 33.1: **Simple Code Using `select()`**

work sockets to which the server is paying attention. Finally, the server calls `select()` to see which of the connections have data available upon them. By then using `FD_ISSET()` in a loop, the event server can see which of the descriptors have data ready and process the incoming data.

Of course, a real server would be more complicated than this, and require logic to use when sending messages, issuing disk I/O, and many other details. For further information, see Stevens and Rago [SR05] for API information, or Pai et. al or Welsh et al. for a good overview of the general flow of event-based servers [PDZ99, WCB01].

33.4 Why Simpler? No Locks Needed

With a single CPU and an event-based application, the problems found in concurrent programs are no longer present. Specifically, because only one event is being handled at a time, there is no need to acquire or release locks; the event-based server cannot be interrupted by another thread because it is decidedly single threaded. Thus, concurrency bugs common in threaded programs do not manifest in the basic event-based approach.

TIP: DON'T BLOCK IN EVENT-BASED SERVERS
Event-based servers enable fine-grained control over scheduling of tasks. However, to maintain such control, no call that blocks the execution the caller can ever be made; failing to obey this design tip will result in a blocked event-based server, frustrated clients, and serious questions as to whether you ever read this part of the book.

33.5 A Problem: Blocking System Calls

Thus far, event-based programming sounds great, right? You program a simple loop, and handle events as they arise. You don't even need to think about locking! But there is an issue: what if an event requires that you issue a system call that might block?

For example, imagine a request comes from a client into a server to read a file from disk and return its contents to the requesting client (much like a simple HTTP request). To service such a request, some event handler will eventually have to issue an `open()` system call to open the file, followed by a series of `read()` calls to read the file. When the file is read into memory, the server will likely start sending the results to the client.

Both the `open()` and `read()` calls may issue I/O requests to the storage system (when the needed metadata or data is not in memory already), and thus may take a long time to service. With a thread-based server, this is no issue: while the thread issuing the I/O request suspends (waiting for the I/O to complete), other threads can run, thus enabling the server to make progress. Indeed, this natural **overlap** of I/O and other computation is what makes thread-based programming quite natural and straightforward.

With an event-based approach, however, there are no other threads to run: just the main event loop. And this implies that if an event handler issues a call that blocks, the *entire* server will do just that: block until the call completes. When the event loop blocks, the system sits idle, and thus is a huge potential waste of resources. We thus have a rule that must be obeyed in event-based systems: no blocking calls are allowed.

33.6 A Solution: Asynchronous I/O

To overcome this limit, many modern operating systems have introduced new ways to issue I/O requests to the disk system, referred to generically as **asynchronous I/O**. These interfaces enable an application to issue an I/O request and return control immediately to the caller, before the I/O has completed; additional interfaces enable an application to determine whether various I/Os have completed.

For example, let us examine the interface provided on Mac OS X (other systems have similar APIs). The APIs revolve around a basic structure,

the `struct aiocb` or **AIO control block** in common terminology. A simplified version of the structure looks like this (see the manual pages for more information):

```
struct aiocb {
    int             aio_fildes;          /* File descriptor */
    off_t           aio_offset;          /* File offset */
    volatile void   *aio_buf;            /* Location of buffer */
    size_t          aio_nbytes;          /* Length of transfer */
};
```

To issue an asynchronous read to a file, an application should first fill in this structure with the relevant information: the file descriptor of the file to be read (`aio_fildes`), the offset within the file (`aio_offset`) as well as the length of the request (`aio_nbytes`), and finally the target memory location into which the results of the read should be copied (`aio_buf`).

After this structure is filled in, the application must issue the asynchronous call to read the file; on Mac OS X, this API is simply the **asynchronous read** API:

```
int aio_read(struct aiocb *aiocbp);
```

This call tries to issue the I/O; if successful, it simply returns right away and the application (i.e., the event-based server) can continue with its work.

There is one last piece of the puzzle we must solve, however. How can we tell when an I/O is complete, and thus that the buffer (pointed to by `aio_buf`) now has the requested data within it?

One last API is needed. On Mac OS X, it is referred to (somewhat confusingly) as `aio_error()`. The API looks like this:

```
int aio_error(const struct aiocb *aiocbp);
```

This system call checks whether the request referred to by `aiocbp` has completed. If it has, the routine returns success (indicated by a zero); if not, EINPROGRESS is returned. Thus, for every outstanding asynchronous I/O, an application can periodically **poll** the system via a call to `aio_error()` to determine whether said I/O has yet completed.

One thing you might have noticed is that it is painful to check whether an I/O has completed; if a program has tens or hundreds of I/Os issued at a given point in time, should it simply keep checking each of them repeatedly, or wait a little while first, or ... ?

To remedy this issue, some systems provide an approach based on the **interrupt**. This method uses UNIX **signals** to inform applications when an asynchronous I/O completes, thus removing the need to repeatedly ask the system. This polling vs. interrupts issue is seen in devices too, as you will see (or already have seen) in the chapter on I/O devices.

ASIDE: **UNIX SIGNALS**

A huge and fascinating infrastructure known as **signals** is present in all modern UNIX variants. At its simplest, signals provide a way to communicate with a process. Specifically, a signal can be delivered to an application; doing so stops the application from whatever it is doing to run a **signal handler**, i.e., some code in the application to handle that signal. When finished, the process just resumes its previous behavior.

Each signal has a name, such as **HUP** (hang up), **INT** (interrupt), **SEGV** (segmentation violation), etc; see the manual page for details. Interestingly, sometimes it is the kernel itself that does the signaling. For example, when your program encounters a segmentation violation, the OS sends it a **SIGSEGV** (prepending **SIG** to signal names is common); if your program is configured to catch that signal, you can actually run some code in response to this erroneous program behavior (which can be useful for debugging). When a signal is sent to a process not configured to handle that signal, some default behavior is enacted; for SEGV, the process is killed.

Here is a simple program that goes into an infinite loop, but has first set up a signal handler to catch SIGHUP:

```
#include <stdio.h>
#include <signal.h>

void handle(int arg) {
    printf("stop wakin' me up...\n");
}

int main(int argc, char *argv[]) {
    signal(SIGHUP, handle);
    while (1)
        ; // doin' nothin' except catchin' some sigs
    return 0;
}
```

You can send signals to it with the **kill** command line tool (yes, this is an odd and aggressive name). Doing so will interrupt the main while loop in the program and run the handler code `handle()`:

```
prompt> ./main &
[3] 36705
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
```

There is a lot more to learn about signals, so much that a single page, much less a single chapter, does not nearly suffice. As always, there is one great source: Stevens and Rago [SR05]. Read more if interested.

In systems without asynchronous I/O, the pure event-based approach cannot be implemented. However, clever researchers have derived methods that work fairly well in their place. For example, Pai et al. [PDZ99] describe a hybrid approach in which events are used to process network packets, and a thread pool is used to manage outstanding I/Os. Read their paper for details.

33.7 Another Problem: State Management

Another issue with the event-based approach is that such code is generally more complicated to write than traditional thread-based code. The reason is as follows: when an event handler issues an asynchronous I/O, it must package up some program state for the next event handler to use when the I/O finally completes; this additional work is not needed in thread-based programs, as the state the program needs is on the stack of the thread. Adya et al. call this work **manual stack management**, and it is fundamental to event-based programming [A+02].

To make this point more concrete, let's look at a simple example in which a thread-based server needs to read from a file descriptor (`fd`) and, once complete, write the data that it read from the file to a network socket descriptor (`sd`). The code (ignoring error checking) looks like this:

```
int rc = read(fd, buffer, size);
rc = write(sd, buffer, size);
```

As you can see, in a multi-threaded program, doing this kind of work is trivial; when the `read()` finally returns, the code immediately knows which socket to write to because that information is on the stack of the thread (in the variable `sd`).

In an event-based system, life is not so easy. To perform the same task, we'd first issue the read asynchronously, using the AIO calls described above. Let's say we then periodically check for completion of the read using the `aio_error()` call; when that call informs us that the read is complete, how does the event-based server know what to do?

The solution, as described by Adya et al. [A+02], is to use an old programming language construct known as a **continuation** [FHK84]. Though it sounds complicated, the idea is rather simple: basically, record the needed information to finish processing this event in some data structure; when the event happens (i.e., when the disk I/O completes), look up the needed information and process the event.

In this specific case, the solution would be to record the socket descriptor (`sd`) in some kind of data structure (e.g., a hash table), indexed by the file descriptor (`fd`). When the disk I/O completes, the event handler would use the file descriptor to look up the continuation, which will return the value of the socket descriptor to the caller. At this point (finally), the server can then do the last bit of work to write the data to the socket.

33.8 What Is Still Difficult With Events

There are a few other difficulties with the event-based approach that we should mention. For example, when systems moved from a single CPU to multiple CPUs, some of the simplicity of the event-based approach disappeared. Specifically, in order to utilize more than one CPU, the event server has to run multiple event handlers in parallel; when doing so, the usual synchronization problems (e.g., critical sections) arise, and the usual solutions (e.g., locks) must be employed. Thus, on modern multicore systems, simple event handling without locks is no longer possible.

Another problem with the event-based approach is that it does not integrate well with certain kinds of systems activity, such as **paging**. For example, if an event-handler page faults, it will block, and thus the server will not make progress until the page fault completes. Even though the server has been structured to avoid *explicit* blocking, this type of *implicit* blocking due to page faults is hard to avoid and thus can lead to large performance problems when prevalent.

A third issue is that event-based code can be hard to manage over time, as the exact semantics of various routines changes [A+02]. For example, if a routine changes from non-blocking to blocking, the event handler that calls that routine must also change to accommodate its new nature, by ripping itself into two pieces. Because blocking is so disastrous for event-based servers, a programmer must always be on the lookout for such changes in the semantics of the APIs each event uses.

Finally, though asynchronous disk I/O is now possible on most platforms, it has taken a long time to get there [PDZ99], and it never quite integrates with asynchronous network I/O in as simple and uniform a manner as you might think. For example, while one would simply like to use the `select()` interface to manage all outstanding I/Os, usually some combination of `select()` for networking and the AIO calls for disk I/O are required.

33.9 Summary

We've presented a bare bones introduction to a different style of concurrency based on events. Event-based servers give control of scheduling to the application itself, but do so at some cost in complexity and difficulty of integration with other aspects of modern systems (e.g., paging). Because of these challenges, no single approach has emerged as best; thus, both threads and events are likely to persist as two different approaches to the same concurrency problem for many years to come. Read some research papers (e.g., [A+02, PDZ99, vB+03, WCB01]) or better yet, write some event-based code, to learn more.

References

[A+02] "Cooperative Task Management Without Manual Stack Management"
Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, John R. Douceur
USENIX ATC '02, Monterey, CA, June 2002
This gem of a paper is the first to clearly articulate some of the difficulties of event-based concurrency, and suggests some simple solutions, as well explores the even crazier idea of combining the two types of concurrency management into a single application!

[FHK84] "Programming With Continuations"
Daniel P. Friedman, Christopher T. Haynes, Eugene E. Kohlbecker
In Program Transformation and Programming Environments, Springer Verlag, 1984
The classic reference to this old idea from the world of programming languages. Now increasingly popular in some modern languages.

[N13] "Node.js Documentation"
By the folks who build node.js
Available: http://nodejs.org/api/
One of the many cool new frameworks that help you readily build web services and applications. Every modern systems hacker should be proficient in frameworks such as this one (and likely, more than one). Spend the time and do some development in one of these worlds and become an expert.

[O96] "Why Threads Are A Bad Idea (for most purposes)"
John Ousterhout
Invited Talk at USENIX '96, San Diego, CA, January 1996
A great talk about how threads aren't a great match for GUI-based applications (but the ideas are more general). Ousterhout formed many of these opinions while he was developing Tcl/Tk, a cool scripting language and toolkit that made it 100x easier to develop GUI-based applications than the state of the art at the time. While the Tk GUI toolkit lives on (in Python for example), Tcl seems to be slowly dying (unfortunately).

[PDZ99] "Flash: An Efficient and Portable Web Server"
Vivek S. Pai, Peter Druschel, Willy Zwaenepoel
USENIX '99, Monterey, CA, June 1999
A pioneering paper on how to structure web servers in the then-burgeoning Internet era. Read it to understand the basics as well as to see the authors' ideas on how to build hybrids when support for asynchronous I/O is lacking.

[SR05] "Advanced Programming in the UNIX Environment"
W. Richard Stevens and Stephen A. Rago
Addison-Wesley, 2005
Once again, we refer to the classic must-have-on-your-bookshelf book of UNIX systems programming. If there is some detail you need to know, it is in here.

[vB+03] "Capriccio: Scalable Threads for Internet Services"
Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, Eric Brewer
SOSP '03, Lake George, New York, October 2003
A paper about how to make threads work at extreme scale; a counter to all the event-based work ongoing at the time.

[WCB01] "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services"
Matt Welsh, David Culler, and Eric Brewer
SOSP '01, Banff, Canada, October 2001
A nice twist on event-based serving that combines threads, queues, and event-based hanlding into one streamlined whole. Some of these ideas have found their way into the infrastructures of companies such as Google, Amazon, and elsewhere.

34

Summary Dialogue on Concurrency

Professor: *So, does your head hurt now?*

Student: *(taking two Motrin tablets) Well, some. It's hard to think about all the ways threads can interleave.*

Professor: *Indeed it is. I am always amazed that when concurrent execution is involved, just a few lines of code can become nearly impossible to understand.*

Student: *Me too! It's kind of embarrassing, as a Computer Scientist, not to be able to make sense of five lines of code.*

Professor: *Oh, don't feel too badly. If you look through the first papers on concurrent algorithms, they are sometimes wrong! And the authors often professors!*

Student: *(gasps) Professors can be ... umm... wrong?*

Professor: *Yes, it is true. Though don't tell anybody — it's one of our trade secrets.*

Student: *I am sworn to secrecy. But if concurrent code is so hard to think about, and so hard to get right, how are we supposed to write correct concurrent code?*

Professor: *Well that is the real question, isn't it? I think it starts with a few simple things. First, keep it simple! Avoid complex interactions between threads, and use well-known and tried-and-true ways to manage thread interactions.*

Student: *Like simple locking, and maybe a producer-consumer queue?*

Professor: *Exactly! Those are common paradigms, and you should be able to produce the working solutions given what you've learned. Second, only use concurrency when absolutely needed; avoid it if at all possible. There is nothing worse than premature optimization of a program.*

Student: *I see — why add threads if you don't need them?*

Professor: *Exactly. Third, if you really need parallelism, seek it in other simplified forms. For example, the Map-Reduce method for writing parallel data analysis code is an excellent example of achieving parallelism without having to handle any of the horrific complexities of locks, condition variables, and the other nasty things we've talked about.*

Student: *Map-Reduce, huh? Sounds interesting — I'll have to read more about it on my own.*

Professor: *Good! You should. In the end, you'll have to do a lot of that, as what we learn together can only serve as the barest introduction to the wealth of knowledge that is out there. Read, read, and read some more! And then try things out, write some code, and then write some more too. As Gladwell talks about in his book "Outliers", you need to put roughly 10,000 hours into something in order to become a real expert. You can't do that all inside of class time!*

Student: *Wow, I'm not sure if that is depressing, or uplifting. But I'll assume the latter, and get to work! Time to write some more concurrent code...*