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 `http://www.barrypopik.com` (on, December 19, 2012, entitled "Tell me and I forget; teach me and I may remember; involve me and I will learn") 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 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.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include "common.h"

int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }
    char *str = argv[1];
    while (1) {
        Spin(1);
        printf("%s\n", str);
    }
    return 0;
}
```

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; by pressing "Control-c" (which on UNIX-based systems will terminate the program running in the foreground) we can halt the program.

```
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
...
```

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

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.

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 a 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 `zsh` shell, which means that the user is able to immediately issue their next command, which in this case is another program to run. If you're using a different shell (e.g., `tcsh`), it works slightly differently; read documentation online for details.

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

int
main(int argc, char *argv[])
{
    int *p = malloc(sizeof(int));                       // a1
    assert(p != NULL);
    printf("(%d) address pointed to by p: %p\n",
           getpid(), p);                                // a2
    *p = 0;                                             // a3
    while (1) {
        Spin(1);
        *p = *p + 1;
        printf("(%d) p: %d\n", getpid(), *p);           // a4
    }
    return 0;
}
```

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) address pointed to by p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

```
prompt> ./mem & ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by 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**

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.

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**.

[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...

2.3 Concurrency

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"
#include "common_threads.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);

    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

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

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.

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` 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 threads threads.c -Wall -pthread
prompt> ./threads 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> ./threads 100000
Initial value : 0
Final value   : 143012    // huh??
prompt> ./threads 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,

[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?

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.

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!

[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.

```c
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    int fd = open("/tmp/file", O_WRONLY|O_CREAT|O_TRUNC,
                  S_IRWXU);
    assert(fd > -1);
    int rc = write(fd, "hello world\n", 13);
    assert(rc == 13);
    close(fd);
    return 0;
}
```

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

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".

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

[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.

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?

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.

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.

[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!

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 challenge — 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 [S68]), 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,

[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...

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 mem-

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, took advantage of the sophisticated GNU tools that already existed [G85], 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). 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.

ory. 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/macOS 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 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.

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" by L. Bairavasundaram, S. Sundararaman, A. Arpaci-Dusseau, R. 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" by 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" by D. Bobrow, J. Burchfiel, D. Murphy, R. 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" by F. Brooks. Addison-Wesley, 1975. *A classic text on software engineering; well worth the read.*

[BOH10] "Computer Systems: A Programmer's Perspective" by R. Bryant and D. 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!*

[G85] "The GNU Manifesto" by R. Stallman. 1985. `www.gnu.org/gnu/manifesto.html`. *A huge part of Linux's success was no doubt the presence of an excellent compiler,* `gcc`*, and other relevant pieces of open software, thanks to the GNU effort headed by Stallman. Stallman is a visionary when it comes to open source, and this manifesto lays out his thoughts as to why.*

[K+61] "One-Level Storage System" by 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" by S. H. Lavington. Communications of the ACM, Volume 21:1, January 1978. *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" by Elliott Organick. MIT Press, 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. 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" by Yale N. Patt, 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" by Dennis M. Ritchie, Ken Thompson. CACM, Volume 17: 7, July 1974. *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" by Scientific Data Systems. TECHNICAL MANUAL, SDS 90 11168, August 1968. *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" by S. Sundararaman, S. Subramanian, A. Rajimwale, A. Arpaci-Dusseau, R. Arpaci-Dusseau, 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.*

Homework

Most (and eventually, all) chapters of this book have homework sections at the end. Doing these homeworks is important, as each lets you, the reader, gain more experience with the concepts presented within the chapter.

There are two types of homeworks. The first is based on **simulation**. A simulation of a computer system is just a simple program that pretends to do some of the interesting parts of what a real system does, and then report some output metrics to show how the system behaves. For example, a hard drive simulator might take a series of requests, simulate how long they would take to get serviced by a hard drive with certain performance characteristics, and then report the average latency of the requests.

The cool thing about simulations is they let you easily explore how systems behave without the difficulty of running a real system. Indeed, they even let you create systems that cannot exist in the real world (for example, a hard drive with unimaginably fast performance), and thus see the potential impact of future technologies.

Of course, simulations are not without their downsides. By their very nature, simulations are just approximations of how a real system behaves. If an important aspect of real-world behavior is omitted, the simulation will report bad results. Thus, results from a simulation should always be treated with some suspicion. In the end, how a system behaves in the real world is what matters.

The second type of homework requires interaction with **real-world code**. Some of these homeworks are measurement focused, whereas others just require some small-scale development and experimentation. Both are just small forays into the larger world you should be getting into, which is how to write systems code in C on UNIX-based systems. Indeed, larger-scale projects, which go beyond these homeworks, are needed to push you in this direction; thus, beyond just doing homeworks, we strongly recommend you do projects to solidify your systems skills. See this page (`https://github.com/remzi-arpacidusseau/ostep-projects`) for some projects.

To do these homeworks, you likely have to be on a UNIX-based machine, running either Linux, macOS, or some similar system. It should also have a C compiler installed (e.g., **gcc**) as well as Python. You should also know how to edit code in a real code editor of some kind.

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 and some high-level intelligence. We call the low-level machinery **mechanisms**; mechanisms are low-level methods or protocols that implement a needed piece of functionality. For example, we'll learn later how to implement a **context**

> TIP: USE TIME SHARING (AND SPACE SHARING)
> **Time sharing** is a basic technique 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 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; once a block is assigned to a file, it is normally not assigned to another file until the user deletes the original file.

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 program will execute next; similarly a **stack pointer** and associated **frame**

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.

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.

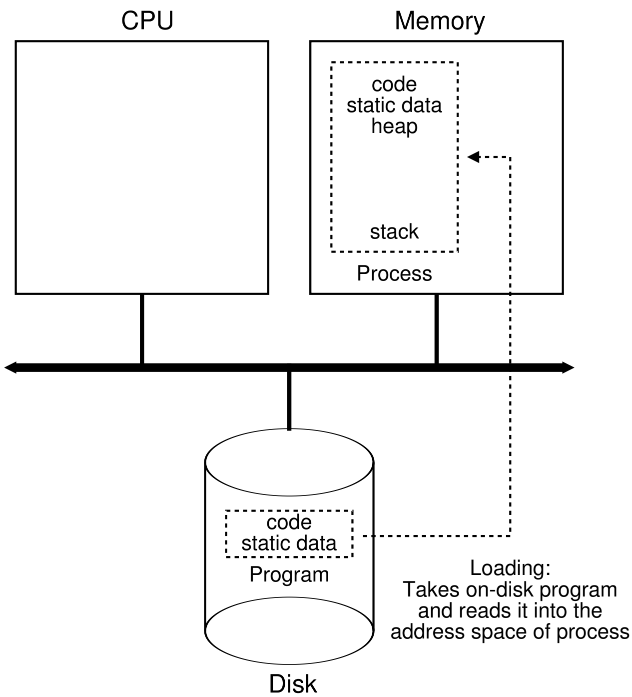


Figure 4.1: **Loading: From Program To Process**

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 and 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.

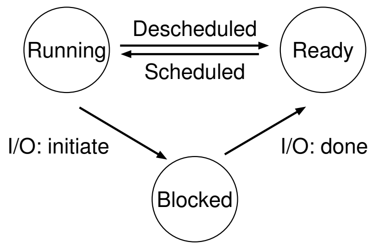


Figure 4.2: **Process: State Transitions**

- **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.

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**

Time	$\textbf{Process}_0$	$\textbf{Process}_1$	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**

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 reading 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 and 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

```
// 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 !zero, sleeping on chan
  int killed;                 // If !zero, has 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**

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.

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

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** (also called the **task 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 (also sometimes called a **process descriptor**).

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.

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.

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

ASIDE: KEY PROCESS TERMS

- The **process** is the major OS abstraction of a running program. At any point in time, the process can be described by its state: the contents of memory in its **address space**, the contents of CPU registers (including the **program counter** and **stack pointer**, among others), and information about I/O (such as open files which can be read or written).

- The **process API** consists of calls programs can make related to processes. Typically, this includes creation, destruction, and other useful calls.

- Processes exist in one of many different **process states**, including running, ready to run, and blocked. Different events (e.g., getting scheduled or descheduled, or waiting for an I/O to complete) transition a process from one of these states to the other.

- A **process list** contains information about all processes in the system. Each entry is found in what is sometimes called a **process control block** (**PCB**), which is really just a structure that contains information about a specific process.

References

[BH70] "The Nucleus of a Multiprogramming System" by Per Brinch Hansen. Communications of the ACM, Volume 13:4, 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" by Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich. From: https://github.com/mit-pdos/xv6-public. *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. We have been using an older version (2012-01-30-1-g1c41342) and hence some examples in the book may not match the latest in the source.*

[DV66] "Programming Semantics for Multiprogrammed Computations" by Jack B. Dennis, 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" by R. Levin, E. Cohen, W. Corwin, F. Pollack, W. Wulf. SOSP '75, Austin, Texas, November 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" by 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 (Simulation)

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 `process-run.py` with the following flags: `-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. Switch the order of the processes: `-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 (`-l 1:0,4:100 -c -S SWITCH_ON_END`), one doing I/O and the other doing CPU work?

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 process 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? (Run `./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: `-s 1 -l 3:50,3:50` or `-s 2 -l 3:50,3:50` or `-s 3 -l 3:50,3:50`. See if you can predict how the trace will turn out. What happens when you use the flag `-I IO_RUN_IMMEDIATE` vs. `-I IO_RUN_LATER`? What happens when you use `-S SWITCH_ON_IO` vs. `-S SWITCH_ON_END`?

5

Interlude: Process API

> ASIDE: INTERLUDES
> Interludes will cover more practical aspects of systems, including a particular focus on operating system APIs and how to use them. If you don't like practical things, you could skip these interludes. But you should like practical things, because, well, they are generally useful in real life; companies, for example, don't usually hire you for your non-practical skills.

In this interlude, we discuss process creation in UNIX systems. UNIX presents one of the most intriguing ways to create a new process with a pair of system calls: `fork()` and `exec()`. A third routine, `wait()`, can be used by a process wishing to wait for a process it has created to complete. We now present these interfaces in more detail, with a few simple examples to motivate us. And thus, our problem:

> CRUX: HOW TO CREATE AND CONTROL PROCESSES
> What interfaces should the OS present for process creation and control? How should these interfaces be designed to enable powerful functionality, ease of use, and high performance?

5.1 The `fork()` System Call

The `fork()` system call is used to create a new process [C63]. However, be forewarned: it is certainly the strangest routine you will ever call[1]. More specifically, you have a running program whose code looks like what you see in Figure 5.1; examine the code, or better yet, type it in and run it yourself!

[1] Well, OK, we admit that we don't know that for sure; who knows what routines you call when no one is looking? But `fork()` is pretty odd, no matter how unusual your routine-calling patterns are.

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

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
  } else {
    // parent goes down this path (main)
    printf("hello, I am parent of %d (pid:%d)\n",
           rc, (int) getpid());
  }
  return 0;
}

```

Figure 5.1: **Calling `fork()` (`p1.c`)**

When you run this program (called `p1.c`), you'll see the following:

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

Let us understand what happened in more detail in `p1.c`. When it first started running, the process prints out a hello world message; included in that message is its **process identifier**, also known as a **PID**. The process has a PID of 29146; in UNIX systems, the PID is used to name the process if one wants to do something with the process, such as (for example) stop it from running. So far, so good.

Now the interesting part begins. The process calls the `fork()` system call, which the OS provides as a way to create a new process. The odd part: the process that is created is an (almost) *exact copy of the calling process*. That means that to the OS, it now looks like there are two copies of the program `p1` running, and both are about to return from the `fork()` system call. The newly-created process (called the **child**, in contrast to the creating **parent**) doesn't start running at `main()`, like you might expect (note, the "hello, world" message only got printed out once); rather, it just comes into life as if it had called `fork()` itself.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {         // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
  } else {              // parent goes down this path (main)
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
           rc, rc_wait, (int) getpid());
  }
  return 0;
}

```

Figure 5.2: **Calling `fork()` And `wait()` (`p2.c`)**

You might have noticed: the child isn't an *exact* copy. Specifically, although it now has its own copy of the address space (i.e., its own private memory), its own registers, its own PC, and so forth, the value it returns to the caller of **fork()** is different. Specifically, while the parent receives the PID of the newly-created child, the child receives a return code of zero. This differentiation is useful, because it is simple then to write the code that handles the two different cases (as above).

You might also have noticed: the output (of `p1.c`) is not **deterministic**. When the child process is created, there are now two active processes in the system that we care about: the parent and the child. Assuming we are running on a system with a single CPU (for simplicity), then either the child or the parent might run at that point. In our example (above), the parent did and thus printed out its message first. In other cases, the opposite might happen, as we show in this output trace:

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

The CPU **scheduler**, a topic we'll discuss in great detail soon, determines which process runs at a given moment in time; because the scheduler is complex, we cannot usually make strong assumptions about what it will choose to do, and hence which process will run first. This **non-determinism**, as it turns out, leads to some interesting problems, particularly in **multi-threaded programs**; hence, we'll see a lot more non-determinism when we study **concurrency** in the second part of the book.

5.2 The `wait()` System Call

So far, we haven't done much: just created a child that prints out a message and exits. Sometimes, as it turns out, it is quite useful for a parent to wait for a child process to finish what it has been doing. This task is accomplished with the `wait()` system call (or its more complete sibling `waitpid()`); see Figure 5.2 for details.

In this example (`p2.c`), the parent process calls `wait()` to delay its execution until the child finishes executing. When the child is done, `wait()` returns to the parent.

Adding a `wait()` call to the code above makes the output deterministic. Can you see why? Go ahead, think about it.

(waiting for you to think .... and done)

Now that you have thought a bit, here is the output:

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (rc_wait:29267) (pid:29266)
prompt>
```

With this code, we now know that the child will always print first. Why do we know that? Well, it might simply run first, as before, and thus print before the parent. However, if the parent does happen to run first, it will immediately call `wait()`; this system call won't return until the child has run and exited[2]. Thus, even when the parent runs first, it politely waits for the child to finish running, then `wait()` returns, and then the parent prints its message.

5.3 Finally, The `exec()` System Call

A final and important piece of the process creation API is the `exec()` system call[3]. This system call is useful when you want to run a program that is different from the calling program. For example, calling `fork()`

[2] There are a few cases where `wait()` returns before the child exits; read the man page for more details, as always. And beware of any absolute and unqualified statements this book makes, such as "the child will always print first" or "UNIX is the best thing in the world, even better than ice cream."

[3] On Linux, there are six variants of `exec()`: `execl`, `execlp()`, `execle()`, `execv()`, `execvp()`, and `execvpe()`. Read the man pages to learn more.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {         // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: "wc" (word count)
    myargs[1] = strdup("p3.c"); // argument: file to count
    myargs[2] = NULL;           // marks end of array
    execvp(myargs[0], myargs);  // runs word count
    printf("this shouldn't print out");
  } else {              // parent goes down this path (main)
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
            rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

Figure 5.3: **Calling `fork()`, `wait()`, And `exec()` (`p3.c`)**

in `p2.c` is only useful if you want to keep running copies of the same program. However, often you want to run a *different* program; `exec()` does just that (Figure 5.3).

In this example, the child process calls `execvp()` in order to run the program `wc`, which is the word counting program. In fact, it runs `wc` on the source file `p3.c`, thus telling us how many lines, words, and bytes are found in the file:

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
      29     107    1030 p3.c
hello, I am parent of 29384 (rc_wait:29384) (pid:29383)
prompt>
```

The `fork()` system call is strange; its partner in crime, `exec()`, is not so normal either. What it does: given the name of an executable (e.g., `wc`), and some arguments (e.g., `p3.c`), it **loads** code (and static data) from that

TIP: GETTING IT RIGHT (LAMPSON'S LAW)

As Lampson states in his well-regarded "Hints for Computer Systems Design" [L83], "**Get it right**. Neither abstraction nor simplicity is a substitute for getting it right." Sometimes, you just have to do the right thing, and when you do, it is way better than the alternatives. There are lots of ways to design APIs for process creation; however, the combination of `fork()` and `exec()` are simple and immensely powerful. Here, the UNIX designers simply got it right. And because Lampson so often "got it right", we name the law in his honor.

executable and overwrites its current code segment (and current static data) with it; the heap and stack and other parts of the memory space of the program are re-initialized. Then the OS simply runs that program, passing in any arguments as the `argv` of that process. Thus, it does *not* create a new process; rather, it transforms the currently running program (formerly `p3`) into a different running program (`wc`). After the `exec()` in the child, it is almost as if `p3.c` never ran; a successful call to `exec()` never returns.

5.4 Why? Motivating The API

Of course, one big question you might have: why would we build such an odd interface to what should be the simple act of creating a new process? Well, as it turns out, the separation of `fork()` and `exec()` is essential in building a UNIX shell, because it lets the shell run code *after* the call to `fork()` but *before* the call to `exec()`; this code can alter the environment of the about-to-be-run program, and thus enables a variety of interesting features to be readily built.

The shell is just a user program[4]. It shows you a **prompt** and then waits for you to type something into it. You then type a command (i.e., the name of an executable program, plus any arguments) into it; in most cases, the shell then figures out where in the file system the executable resides, calls `fork()` to create a new child process to run the command, calls some variant of `exec()` to run the command, and then waits for the command to complete by calling `wait()`. When the child completes, the shell returns from `wait()` and prints out a prompt again, ready for your next command.

The separation of `fork()` and `exec()` allows the shell to do a whole bunch of useful things rather easily. For example:

```
prompt> wc p3.c > newfile.txt
```

[4]And there are lots of shells; `tcsh`, `bash`, and `zsh` to name a few. You should pick one, read its man pages, and learn more about it; all UNIX experts do.

In the example above, the output of the program `wc` is **redirected** into the output file `newfile.txt` (the greater-than sign is how said redirection is indicated). The way the shell accomplishes this task is quite simple: when the child is created, before calling `exec()`, the shell closes **standard output** and opens the file `newfile.txt`. By doing so, any output from the soon-to-be-running program `wc` are sent to the file instead of the screen.

Figure 5.4 (page 8) shows a program that does exactly this. The reason this redirection works is due to an assumption about how the operating system manages file descriptors. Specifically, UNIX systems start looking for free file descriptors at zero. In this case, STDOUT_FILENO will be the first available one and thus get assigned when `open()` is called. Subsequent writes by the child process to the standard output file descriptor, for example by routines such as `printf()`, will then be routed transparently to the newly-opened file instead of the screen.

Here is the output of running the `p4.c` program:

```
prompt> ./p4
prompt> cat p4.output
      32     109     846 p4.c
prompt>
```

You'll notice (at least) two interesting tidbits about this output. First, when `p4` is run, it looks as if nothing has happened; the shell just prints the command prompt and is immediately ready for your next command. However, that is not the case; the program `p4` did indeed call `fork()` to create a new child, and then run the `wc` program via a call to `execvp()`. You don't see any output printed to the screen because it has been redirected to the file `p4.output`. Second, you can see that when we `cat` the output file, all the expected output from running `wc` is found. Cool, right?

UNIX pipes are implemented in a similar way, but with the `pipe()` system call. In this case, the output of one process is connected to an in-kernel **pipe** (i.e., queue), and the input of another process is connected to that same pipe; thus, the output of one process seamlessly is used as input to the next, and long and useful chains of commands can be strung together. As a simple example, consider looking for a word in a file, and then counting how many times said word occurs; with pipes and the utilities `grep` and `wc`, it is easy; just type `grep -o foo file | wc -l` into the command prompt and marvel at the result.

Finally, while we just have sketched out the process API at a high level, there is a lot more detail about these calls out there to be learned and digested; we'll learn more, for example, about file descriptors when we talk about file systems in the third part of the book. For now, suffice it to say that the `fork()/exec()` combination is a powerful way to create and manipulate processes.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child: redirect standard output to a file
    close(STDOUT_FILENO);
    open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

    // now exec "wc"...
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: wc (word count)
    myargs[1] = strdup("p4.c"); // arg: file to count
    myargs[2] = NULL;           // mark end of array
    execvp(myargs[0], myargs);  // runs word count
  } else {
    // parent goes down this path (main)
    int rc_wait = wait(NULL);
  }
  return 0;
}
```

Figure 5.4: **All Of The Above With Redirection (p4.c)**

5.5 Process Control And Users

Beyond `fork()`, `exec()`, and `wait()`, there are a lot of other interfaces for interacting with processes in UNIX systems. For example, the `kill()` system call is used to send **signals** to a process, including directives to pause, die, and other useful imperatives. For convenience, in most UNIX shells, certain keystroke combinations are configured to deliver a specific signal to the currently running process; for example, control-c sends a `SIGINT` (interrupt) to the process (normally terminating it) and control-z sends a `SIGTSTP` (stop) signal thus pausing the process in mid-execution (you can resume it later with a command, e.g., the `fg` built-in command found in many shells).

The entire signals subsystem provides a rich infrastructure to deliver external events to processes, including ways to receive and process those signals within individual processes, and ways to send signals to individual processes as well as entire **process groups**. To use this form of com-

ASIDE: RTFM — READ THE MAN PAGES

Many times in this book, when referring to a particular system call or library call, we'll tell you to read the **manual pages**, or **man pages** for short. Man pages are the original form of documentation that exist on UNIX systems; realize that they were created before the thing called **the web** existed.

Spending some time reading man pages is a key step in the growth of a systems programmer; there are tons of useful tidbits hidden in those pages. Some particularly useful pages to read are the man pages for whichever shell you are using (e.g., **tcsh**, or **bash**), and certainly for any system calls your program makes (in order to see what return values and error conditions exist).

Finally, reading the man pages can save you some embarrassment. When you ask colleagues about some intricacy of `fork()`, they may simply reply: "RTFM." This is your colleagues' way of gently urging you to Read The Man pages. The F in RTFM just adds a little color to the phrase...

munication, a process should use the `signal()` system call to "catch" various signals; doing so ensures that when a particular signal is delivered to a process, it will suspend its normal execution and run a particular piece of code in response to the signal. Read elsewhere [SR05] to learn more about signals and their many intricacies.

This naturally raises the question: who can send a signal to a process, and who cannot? Generally, the systems we use can have multiple people using them at the same time; if one of these people can arbitrarily send signals such as `SIGINT` (to interrupt a process, likely terminating it), the usability and security of the system will be compromised. As a result, modern systems include a strong conception of the notion of a **user**. The user, after entering a password to establish credentials, logs in to gain access to system resources. The user may then launch one or many processes, and exercise full control over them (pause them, kill them, etc.). Users generally can only control their own processes; it is the job of the operating system to parcel out resources (such as CPU, memory, and disk) to each user (and their processes) to meet overall system goals.

5.6 Useful Tools

There are many command-line tools that are useful as well. For example, using the `ps` command allows you to see which processes are running; read the **man pages** for some useful flags to pass to `ps`. The tool `top` is also quite helpful, as it displays the processes of the system and how much CPU and other resources they are eating up. Humorously, many times when you run it, `top` claims it is the top resource hog; perhaps it is a bit of an egomaniac. The command `kill` can be used to send arbitrary

ASIDE: THE SUPERUSER (ROOT)

A system generally needs a user who can **administer** the system, and is not limited in the way most users are. Such a user should be able to kill an arbitrary process (e.g., if it is abusing the system in some way), even though that process was not started by this user. Such a user should also be able to run powerful commands such as `shutdown` (which, unsurprisingly, shuts down the system). In UNIX-based systems, these special abilities are given to the **superuser** (sometimes called **root**). While most users can't kill other users processes, the superuser can. Being root is much like being Spider-Man: with great power comes great responsibility [QI15]. Thus, to increase **security** (and avoid costly mistakes), it's usually better to be a regular user; if you do need to be root, tread carefully, as all of the destructive powers of the computing world are now at your fingertips.

signals to processes, as can the slightly more user friendly `killall`. Be sure to use these carefully; if you accidentally kill your window manager, the computer you are sitting in front of may become quite difficult to use.

Finally, there are many different kinds of CPU meters you can use to get a quick glance understanding of the load on your system; for example, we always keep **MenuMeters** (from Raging Menace software) running on our Macintosh toolbars, so we can see how much CPU is being utilized at any moment in time. In general, the more information about what is going on, the better.

5.7 Summary

We have introduced some of the APIs dealing with UNIX process creation: `fork()`, `exec()`, and `wait()`. However, we have just skimmed the surface. For more detail, read Stevens and Rago [SR05], of course, particularly the chapters on Process Control, Process Relationships, and Signals; there is much to extract from the wisdom therein.

While our passion for the UNIX process API remains strong, we should also note that such positivity is not uniform. For example, a recent paper by systems researchers from Microsoft, Boston University, and ETH in Switzerland details some problems with `fork()`, and advocates for other, simpler process creation APIs such as **`spawn()`** [B+19]. Read it, and the related work it refers to, to understand this different vantage point. While it's generally good to trust this book, remember too that the authors have opinions; those opinions may not (always) be as widely shared as you might think.

ASIDE: KEY PROCESS API TERMS

- Each process has a name; in most systems, that name is a number known as a **process ID** (**PID**).
- The **fork()** system call is used in UNIX systems to create a new process. The creator is called the **parent**; the newly created process is called the **child**. As sometimes occurs in real life [J16], the child process is a nearly identical copy of the parent.
- The **wait()** system call allows a parent to wait for its child to complete execution.
- The **exec()** family of system calls allows a child to break free from its similarity to its parent and execute an entirely new program.
- A UNIX **shell** commonly uses `fork()`, `wait()`, and `exec()` to launch user commands; the separation of fork and exec enables features like **input/output redirection**, **pipes**, and other cool features, all without changing anything about the programs being run.
- Process control is available in the form of **signals**, which can cause jobs to stop, continue, or even terminate.
- Which processes can be controlled by a particular person is encapsulated in the notion of a **user**; the operating system allows multiple users onto the system, and ensures users can only control their own processes.
- A **superuser** can control all processes (and indeed do many other things); this role should be assumed infrequently and with caution for security reasons.

References

[B+19] "A fork() in the road" by Andrew Baumann, Jonathan Appavoo, Orran Krieger, Timothy Roscoe. HotOS '19, Bertinoro, Italy. *A fun paper full of `fork()`ing rage. Read it to get an opposing viewpoint on the* UNIX *process API. Presented at the always lively HotOS workshop, where systems researchers go to present extreme opinions in the hopes of pushing the community in new directions.*

[C63] "A Multiprocessor System Design" by Melvin E. Conway. AFIPS '63 Fall Joint Computer Conference, New York, USA 1963. *An early paper on how to design multiprocessing systems; may be the first place the term `fork()` was used in the discussion of spawning new processes.*

[DV66] "Programming Semantics for Multiprogrammed Computations" by Jack B. Dennis and Earl C. Van Horn. Communications of the ACM, Volume 9, Number 3, March 1966. *A classic paper that outlines the basics of multiprogrammed computer systems. Undoubtedly had great influence on Project MAC, Multics, and eventually* UNIX.

[J16] "They could be twins!" by Phoebe Jackson-Edwards. The Daily Mail. March 1, 2016.. *This hard-hitting piece of journalism shows a bunch of weirdly similar child/parent photos and is frankly kind of mesmerizing. Go ahead, waste two minutes of your life and check it out. But don't forget to come back here! This, in a microcosm, is the danger of surfing the web.*

[L83] "Hints for Computer Systems Design" by Butler Lampson. ACM Operating Systems Review, Volume 15:5, October 1983. *Lampson's famous hints on how to design computer systems. You should read it at some point in your life, and probably at many points in your life.*

[QI15] "With Great Power Comes Great Responsibility" by The Quote Investigator. Available: `https://quoteinvestigator.com/2015/07/23/great-power`. *The quote investigator concludes that the earliest mention of this concept is 1793, in a collection of decrees made at the French National Convention. The specific quote: "Ils doivent envisager qu'une grande responsabilité est la suite inséparable d'un grand pouvoir", which roughly translates to "They must consider that great responsibility follows inseparably from great power." Only in 1962 did the following words appear in Spider-Man: "...with great power there must also come–great responsibility!" So it looks like the French Revolution gets credit for this one, not Stan Lee. Sorry, Stan.*

[SR05] "Advanced Programming in the UNIX Environment" by W. Richard Stevens, Stephen A. Rago. Addison-Wesley, 2005. *All nuances and subtleties of using* UNIX *APIs are found herein. Buy this book! Read it! And most importantly, **live it.***

Homework (Simulation)

This simulation homework focuses on `fork.py`, a simple process creation simulator that shows how processes are related in a single "familial" tree. Read the relevant README for details about how to run the simulator.

Questions

1. Run `./fork.py -s 10` and see which actions are taken. Can you predict what the process tree looks like at each step? Use the `-c` flag to check your answers. Try some different random seeds (`-s`) or add more actions (`-a`) to get the hang of it.

2. One control the simulator gives you is the `fork_percentage`, controlled by the `-f` flag. The higher it is, the more likely the next action is a fork; the lower it is, the more likely the action is an exit. Run the simulator with a large number of actions (e.g., `-a 100`) and vary the `fork_percentage` from 0.1 to 0.9. What do you think the resulting final process trees will look like as the percentage changes? Check your answer with `-c`.

3. Now, switch the output by using the `-t` flag (e.g., run `./fork.py -t`). Given a set of process trees, can you tell which actions were taken?

4. One interesting thing to note is what happens when a child exits; what happens to its children in the process tree? To study this, let's create a specific example: `./fork.py -A a+b,b+c,c+d,c+e,c-`. This example has process 'a' create 'b', which in turn creates 'c', which then creates 'd' and 'e'. However, then, 'c' exits. What do you think the process tree should like after the exit? What if you use the `-R` flag? Learn more about what happens to orphaned processes on your own to add more context.

5. One last flag to explore is the `-F` flag, which skips intermediate steps and only asks to fill in the final process tree. Run `./fork.py -F` and see if you can write down the final tree by looking at the series of actions generated. Use different random seeds to try this a few times.

6. Finally, use both `-t` and `-F` together. This shows the final process tree, but then asks you to fill in the actions that took place. By looking at the tree, can you determine the exact actions that took place? In which cases can you tell? In which can't you tell? Try some different random seeds to delve into this question.

ASIDE: CODING HOMEWORKS

Coding homeworks are small exercises where you write code to run on a real machine to get some experience with some basic operating system APIs. After all, you are (probably) a computer scientist, and therefore should like to code, right? If you don't, there is always CS theory, but that's pretty hard. Of course, to truly become an expert, you have to spend more than a little time hacking away at the machine; indeed, find every excuse you can to write some code and see how it works. Spend the time, and become the wise master you know you can be.

Homework (Code)

In this homework, you are to gain some familiarity with the process management APIs about which you just read. Don't worry – it's even more fun than it sounds! You'll in general be much better off if you find as much time as you can to write some code, so why not start now?

Questions

1. Write a program that calls `fork()`. Before calling `fork()`, have the main process access a variable (e.g., `x`) and set its value to something (e.g., `100`). What value is the variable in the child process? What happens to the variable when both the child and parent change the value of `x`?

2. Write a program that opens a file (with the `open()` system call) and then calls `fork()` to create a new process. Can both the child and parent access the file descriptor returned by `open()`? What happens when they are writing to the file concurrently, i.e., at the same time?

3. Write another program using `fork()`. The child process should print "hello"; the parent process should print "goodbye". You should try to ensure that the child process always prints first; can you do this *without* calling **wait()** in the parent?

4. Write a program that calls `fork()` and then calls some form of `exec()` to run the program `/bin/ls`. See if you can try all of the variants of `exec()`, including (on Linux) `execl()`, `execle()`, `execlp()`, `execv()`, `execvp()`, and `execvpe()`. Why do you think there are so many variants of the same basic call?

5. Now write a program that uses `wait()` to wait for the child process to finish in the parent. What does `wait()` return? What happens if you use `wait()` in the child?

6. Write a slight modification of the previous program, this time using `waitpid()` instead of `wait()`. When would `waitpid()` be useful?

7. Write a program that creates a child process, and then in the child closes standard output (`STDOUT_FILENO`). What happens if the child calls `printf()` to print some output after closing the descriptor?

8. Write a program that creates two children, and connects the standard output of one to the standard input of the other, using the `pipe()` system call.

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 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

[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**

locations of these **trap handlers**, usually with some kind of special instruction. 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.

TIP: BE WARY OF USER INPUTS IN SECURE SYSTEMS

Even though we have taken great pains to protect the OS during system calls (by adding a hardware trapping mechanism, and ensuring all calls to the OS are routed through it), there are still many other aspects to implementing a **secure** operating system that we must consider. One of these is the handling of arguments at the system call boundary; the OS must check what the user passes in and ensure that arguments are properly specified, or otherwise reject the call.

For example, with a `write()` system call, the user specifies an address of a buffer as a source of the write call. If the user (either accidentally or maliciously) passes in a "bad" address (e.g., one inside the kernel's portion of the address space), the OS must detect this and reject the call. Otherwise, it would be possible for a user to read all of kernel memory; given that kernel (virtual) memory also usually includes all of the physical memory of the system, this small slip would enable a program to read the memory of any other process in the system.

In general, a secure system must treat user inputs with great suspicion. Not doing so will undoubtedly lead to easily hacked software, a despairing sense that the world is an unsafe and scary place, and the loss of job security for the all-too-trusting OS developer.

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 limited direct execution (**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, 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?

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

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?

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 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, and 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,

> 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.

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!

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 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.

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) → k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call `switch()` routine save regs(A) → proc_t(A) restore regs(B) ← proc_t(B) switch to k-stack(B) **return-from-trap (into B)**		
	restore regs(B) ← k-stack(B) move to user mode jump to B's PC	
		Process B ...

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

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 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

```
# void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
  # Save old registers
  movl 4(%esp), %eax  # put old ptr into eax
  popl 0(%eax)        # save the old IP
  movl %esp, 4(%eax)  # and stack
  movl %ebx, 8(%eax)  # and other registers
  movl %ecx, 12(%eax)
  movl %edx, 16(%eax)
  movl %esi, 20(%eax)
  movl %edi, 24(%eax)
  movl %ebp, 28(%eax)

  # Load new registers
  movl 4(%esp), %eax  # put new ptr into eax
  movl 28(%eax), %ebp # restore other registers
  movl 24(%eax), %edi
  movl 20(%eax), %esi
  movl 16(%eax), %edx
  movl 12(%eax), %ecx
  movl 8(%eax), %ebx
  movl 4(%eax), %esp  # stack is switched here
  pushl 0(%eax)       # return addr put in place
  ret                 # finally return into new ctxt
```

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

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.

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.

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.

ASIDE: KEY CPU VIRTUALIZATION TERMS (MECHANISMS)

- The CPU should support at least two modes of execution: a restricted **user mode** and a privileged (non-restricted) **kernel mode**.
- Typical user applications run in user mode, and use a **system call** to **trap** into the kernel to request operating system services.
- The trap instruction saves register state carefully, changes the hardware status to kernel mode, and jumps into the OS to a pre-specified destination: the **trap table**.
- When the OS finishes servicing a system call, it returns to the user program via another special **return-from-trap** instruction, which reduces privilege and returns control to the instruction after the trap that jumped into the OS.
- The trap tables must be set up by the OS at boot time, and make sure that they cannot be readily modified by user programs. All of this is part of the **limited direct execution** protocol which runs programs efficiently but without loss of OS control.
- Once a program is running, the OS must use hardware mechanisms to ensure the user program does not run forever, namely the **timer interrupt**. This approach is a **non-cooperative** approach to CPU scheduling.
- Sometimes the OS, during a timer interrupt or system call, might wish to switch from running the current process to a different one, a low-level technique known as a **context switch**.

.

References

[A79] "Alto User's Handbook" by Xerox. 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" by G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, A. 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" by Volume 3A and 3B: System Programming Guide. Intel Corporation, January 2011. *This is just a boring manual, but sometimes those are useful.*

[K+61] "One-Level Storage System" by 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" by 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" by 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" by 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" by Apple Computer, Inc.. January 2011. `http://en.wikipedia.org/wiki/Mac_OS_9` . *You can probably even find an OS 9 emulator out there if you want to; check it out, it's a fun little Mac!*

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

[P10] "The Single UNIX Specification, Version 3" by 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. Like, unless someone is paying you to read it. Or, you're just so curious you can't help it!*

[S07] "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)" by 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?

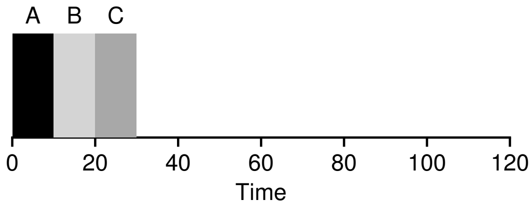


Figure 7.1: **FIFO Simple Example**

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.

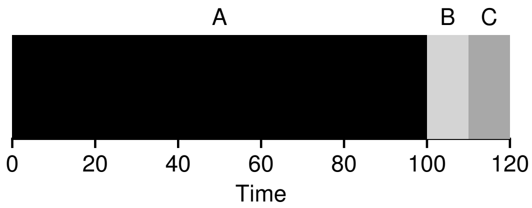


Figure 7.2: **Why FIFO Is Not That Great**

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

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.

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 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.

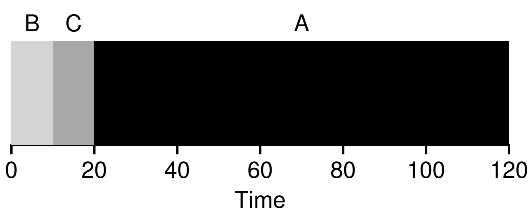


Figure 7.3: **SJF Simple Example**

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.

[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.

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

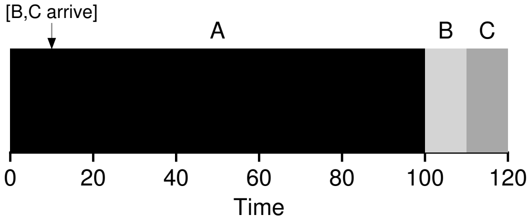


Figure 7.4: **SJF With Late Arrivals From B and C**

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

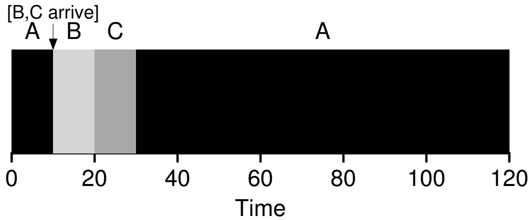


Figure 7.5: **STCF Simple Example**

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.

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**.

We define response time as the time from when the job arrives in a system to the first time it is scheduled[3]. More formally:

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

[3]Some define it slightly differently, e.g., to also include the time until the job produces some kind of "response"; our definition is the best-case version of this, essentially assuming that the job produces a response instantaneously.

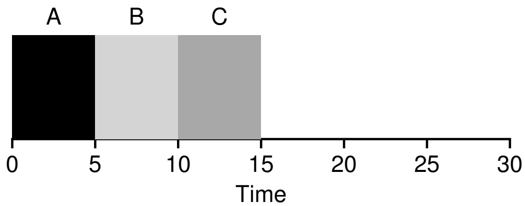


Figure 7.6: **SJF Again (Bad for Response Time)**

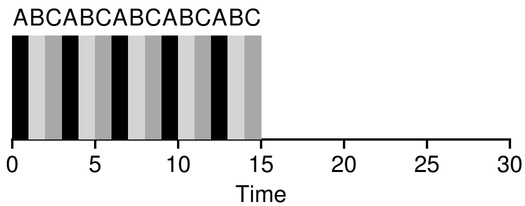


Figure 7.7: **Round Robin (Good For Response Time)**

For example, if we had the schedule from Figure 7.5 (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 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

> 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.

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.

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,

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.

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[4].

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.

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).

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,

[4] 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.*

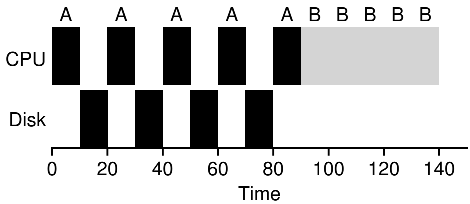


Figure 7.8: **Poor Use Of Resources**

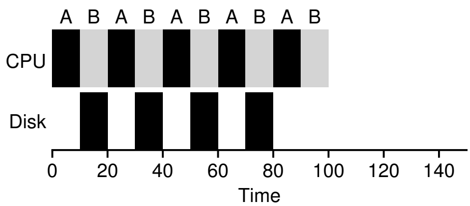


Figure 7.9: **Overlap Allows Better Use Of Resources**

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 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" by 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" by 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" by 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" by 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" by 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" by 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" by Thomas E. Phipps Jr., 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" by Jeffrey C. Mogul, 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" by Authors: Unknown.. Wikipedia (as of December 2015). `http://en.wikipedia.org/wiki/You_can't_have_your_cake_and_eat_it`. *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 (Simulation)

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, page 3). 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-

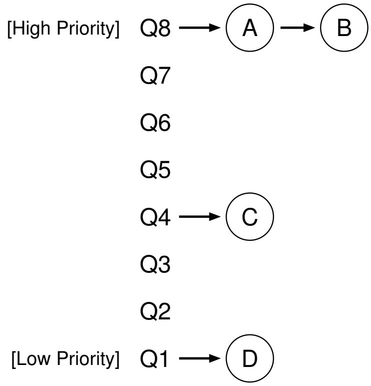


Figure 8.1: **MLFQ Example**

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.

For this, we need a new concept, which we will call the job's **allotment**. The allotment is the amount of time a job can spend at a given priority level before the scheduler reduces its priority. For simplicity, at first, we will assume the allotment is equal to a single time slice.

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 its allotment while running, its priority is *reduced* (i.e., it moves down one queue).
- **Rule 4b:** If a job gives up the CPU (for example, by performing an I/O operation) before the allotment is up, it stays at the *same* priority level (i.e., its allotment is reset).

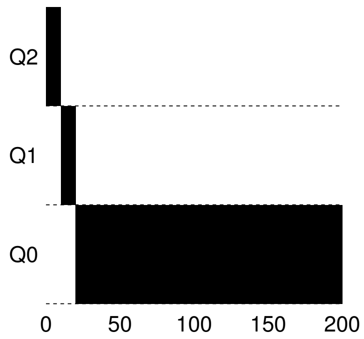


Figure 8.2: **Long-running Job Over Time**

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, with a time slice of 10 ms (and with the allotment set equal to the time slice). Figure 8.2 shows what happens to this job over time in a three-queue scheduler.

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 on page 5 (left) plots the results of this scenario. Job 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 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.

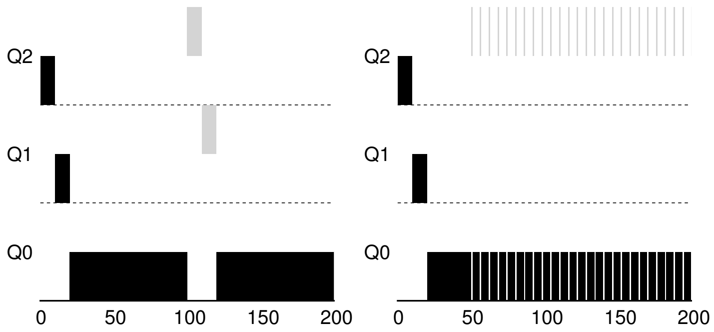


Figure 8.3: **Along Came An Interactive Job: Two Examples**

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 allotment, 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 allotment is complete; in such case, we don't wish to penalize the job and thus simply keep it at the same level.

Figure 8.3 (right) 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)

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

TIP: SCHEDULING MUST BE SECURE FROM ATTACK

You might think that a scheduling policy, whether inside the OS itself (as discussed herein), or in a broader context (e.g., in a distributed storage system's I/O request handling [Y+18]), is not a **security** concern, but in increasingly many cases, it is exactly that. Consider the modern datacenter, in which users from around the world share CPUs, memories, networks, and storage systems; without care in policy design and enforcement, a single user may be able to adversely harm others and gain advantage for itself. Thus, scheduling policy forms an important part of the security of a system, and should be carefully constructed.

the following attack: before the allotment is used, issue an I/O operation (e.g., to a file) 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 the allotment 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 the 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 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 8.4 (page 7). 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 100 ms (which is likely too small of a value, but used here for the example),

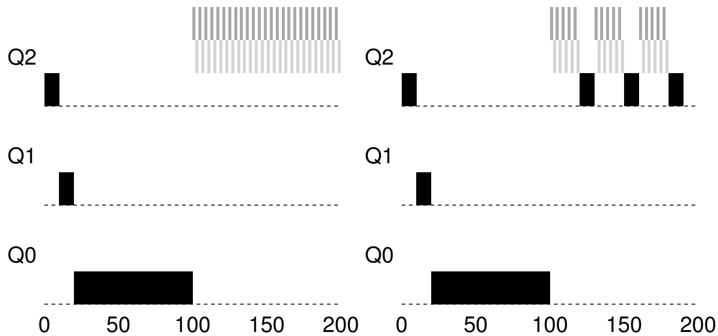


Figure 8.4: **Without (Left) and With (Right) Priority Boost**

and thus we at least guarantee that the long-running job will make some progress, getting boosted to the highest priority every 100 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. As such, it is often left to the system administrator to find the right value – or in the modern world, increasingly, to automatic methods based on machine learning [A+17].

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 its allotment expires. So what should we do?

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**.

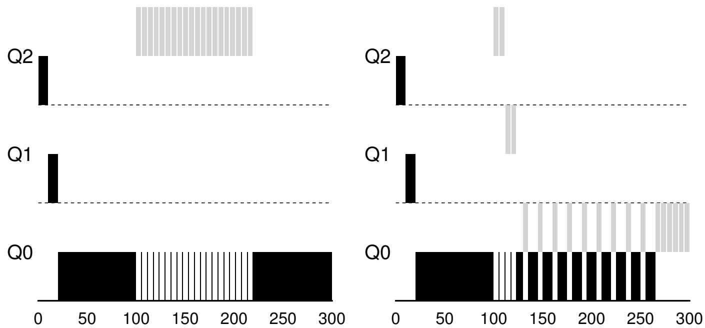


Figure 8.5: **Without (Left) and With (Right) Gaming Tolerance**

The solution here is to perform better **accounting** of CPU time at each level of the MLFQ. Instead of forgetting how much of its allotment a process used at a given level when it performs I/O, the scheduler should keep track; once a process has used its allotment, it is demoted to the next priority queue. Whether it uses its allotment in one long burst or many small ones should 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 8.5 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 before its allotment ends, thus staying at the same priority level, and dominating CPU time. With better accounting in place (right), 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? The allotment? 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

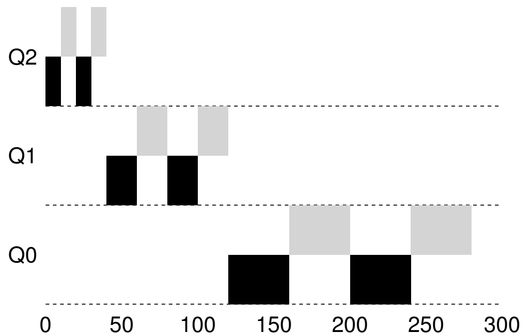


Figure 8.6: **Lower Priority, Longer Quanta**

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 8.6 shows an example in which two jobs run for 20 ms at the highest queue (with a 10-ms time slice), 40 ms in the middle (20-ms time slice), and with a 40-ms time slice at the lowest.

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.

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]).

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.

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 round-robin fashion using the time slice (quantum length) of the given queue.
- **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

[A+17] "Automatic Database Management System Tuning Through Large-scale Machine Learning" by Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, Bohan Zhang. SIGMOD '17. *This isn't about the application of machine learning to CPU scheduling in the OS, but rather a cool early example of automatically tuning parameters of a database via ML techniques. Worth a read, if you like ML... which, alas, everyone seems to these days.*

[AD00] "Multilevel Feedback Queue Scheduling in Solaris" by Andrea Arpaci-Dusseau. Available: http://www.ostep.org/Citations/notes-solaris.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" by 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" by 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" by 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" by 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" by 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" by Richard McDougall. Prentice-Hall, 2006. *A good book about Solaris and how it works.*

[O11] "John Ousterhout's Home Page" by John Ousterhout. `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" by R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, J. Zelenka. SOSP '95, Copper Mountain, Colorado, October 1995. *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.*

[Y+18] "Principled Schedulability Analysis for Distributed Storage Systems using Thread Architecture Models" by Suli Yang, Jing Liu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. OSDI '18, San Diego, California. *A recent work of our group that demonstrates the difficulty of scheduling I/O requests within modern distributed storage systems such as Hive/HDFS, Cassandra, MongoDB, and Riak. Without care, a single user might be able to monopolize system resources.*

Homework (Simulation)

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 early 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 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 have 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 attains worst-case performance 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 12
```

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 A's currency. User B is running only 1 job and gives it 10 tickets (out of 10 total). The system converts 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 is converted to 100 tickets. The lottery is then 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.

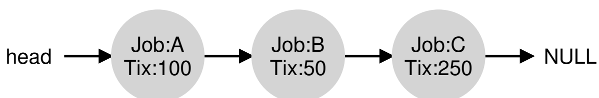
```
// counter: used to track if we've found the winner yet
int counter = 0;

// winner: use some call to a random number generator to
//         get a value, between 0 and the total # of tickets
int winner = getrandom(0, totaltickets);

// current: use this to walk through the list of jobs
node_t *current = head;
while (current) {
    counter = counter + current->tickets;
    if (counter > winner)
        break; // found the winner
    current = current->next;
}
// '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).

[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`.

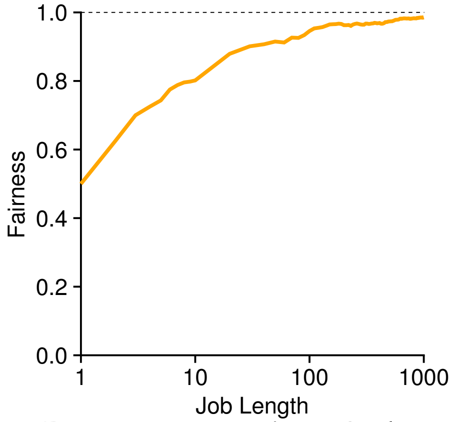


Figure 9.2: **Lottery Fairness Study**

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 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 **fairness metric**, F 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), $F = \frac{10}{20} = 0.5$. When both jobs finish at nearly the same time, F will be quite close to 1. In this scenario, that is our goal: a perfectly fair scheduler would achieve $F = 1$.

Figure 9.2 plots the average fairness 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 fairness can be quite low. Only as the jobs run for a significant number of time slices does the lottery scheduler approach the desired fair 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 Stride Scheduling

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]:

```
curr = remove_min(queue);   // pick client with min pass
schedule(curr);             // run for quantum
curr->pass += curr->stride; // update pass using stride
insert(queue, curr);        // return curr to 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

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 it, updating its pass to 80 (C's stride is 40, as you recall). Then C will 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 The Linux Completely Fair Scheduler (CFS)

Despite these earlier works in fair-share scheduling, the current Linux approach achieves similar goals in an alternate manner. The scheduler, entitled the **Completely Fair Scheduler** (or **CFS**) [J09], implements fair-share scheduling, but does so in a highly efficient and scalable manner.

To achieve its efficiency goals, CFS aims to spend very little time making scheduling decisions, through both its inherent design and its clever use of data structures well-suited to the task. Recent studies have shown that scheduler efficiency is surprisingly important; specifically, in a study of Google datacenters, Kanev et al. show that even after aggressive optimization, scheduling uses about 5% of overall datacenter CPU time. Reducing that overhead as much as possible is thus a key goal in modern scheduler architecture.

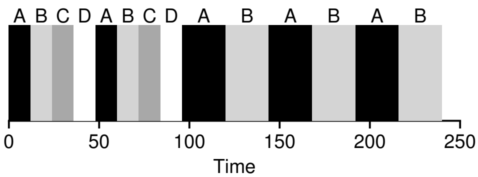


Figure 9.4: **CFS Simple Example**

Basic Operation

Whereas most schedulers are based around the concept of a fixed time slice, CFS operates a bit differently. Its goal is simple: to fairly divide a CPU evenly among all competing processes. It does so through a simple counting-based technique known as **virtual runtime** (**`vruntime`**).

As each process runs, it accumulates `vruntime`. In the most basic case, each process's `vruntime` increases at the same rate, in proportion with physical (real) time. When a scheduling decision occurs, CFS will pick the process with the *lowest* `vruntime` to run next.

This raises a question: how does the scheduler know when to stop the currently running process, and run the next one? The tension here is clear: if CFS switches too often, fairness is increased, as CFS will ensure that each process receives its share of CPU even over miniscule time windows, but at the cost of performance (too much context switching); if CFS switches less often, performance is increased (reduced context switching), but at the cost of near-term fairness.

CFS manages this tension through various control parameters. The first is **`sched_latency`**. CFS uses this value to determine how long one process should run before considering a switch (effectively determining its time slice but in a dynamic fashion). A typical `sched_latency` value is 48 (milliseconds); CFS divides this value by the number (n) of processes running on the CPU to determine the time slice for a process, and thus ensures that over this period of time, CFS will be completely fair.

For example, if there are $n = 4$ processes running, CFS divides the value of `sched_latency` by n to arrive at a per-process time slice of 12 ms. CFS then schedules the first job and runs it until it has used 12 ms of (virtual) runtime, and then checks to see if there is a job with lower `vruntime` to run instead. In this case, there is, and CFS would switch to one of the three other jobs, and so forth. Figure 9.4 shows an example where the four jobs (A, B, C, D) each run for two time slices in this fashion; two of them (C, D) then complete, leaving just two remaining, which then each run for 24 ms in round-robin fashion.

But what if there are "too many" processes running? Wouldn't that lead to too small of a time slice, and thus too many context switches? Good question! And the answer is yes.

To address this issue, CFS adds another parameter, **`min_granularity`**, which is usually set to a value like 6 ms. CFS will never set the time slice of a process to less than this value, ensuring that not too much time is spent in scheduling overhead.

For example, if there are ten processes running, our original calculation would divide `sched_latency` by ten to determine the time slice (result: 4.8 ms). However, because of `min_granularity`, CFS will set the time slice of each process to 6 ms instead. Although CFS won't (quite) be perfectly fair over the target scheduling latency (`sched_latency`) of 48 ms, it will be close, while still achieving high CPU efficiency.

Note that CFS utilizes a periodic timer interrupt, which means it can only make decisions at fixed time intervals. This interrupt goes off frequently (e.g., every 1 ms), giving CFS a chance to wake up and determine if the current job has reached the end of its run. If a job has a time slice that is not a perfect multiple of the timer interrupt interval, that is OK; CFS tracks `vruntime` precisely, which means that over the long haul, it will eventually approximate ideal sharing of the CPU.

Weighting (Niceness)

CFS also enables controls over process priority, enabling users or administrators to give some processes a higher share of the CPU. It does this not with tickets, but through a classic UNIX mechanism known as the **nice** level of a process. The nice parameter can be set anywhere from -20 to +19 for a process, with a default of 0. Positive nice values imply *lower* priority and negative values imply *higher* priority; when you're too nice, you just don't get as much (scheduling) attention, alas.

CFS maps the nice value of each process to a `weight`, as shown here:

```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */  9548,  7620,  6100,  4904,  3906,
    /*  -5 */  3121,  2501,  1991,  1586,  1277,
    /*   0 */  1024,   820,   655,   526,   423,
    /*   5 */   335,   272,   215,   172,   137,
    /*  10 */   110,    87,    70,    56,    45,
    /*  15 */    36,    29,    23,    18,    15,
};
```

These weights allow us to compute the effective time slice of each process (as we did before), but now accounting for their priority differences. The formula used to do so is as follows, assuming n processes:

$$\texttt{time\_slice}_k = \frac{\texttt{weight}_k}{\sum_{i=0}^{n-1} \texttt{weight}_i} \cdot \texttt{sched\_latency} \tag{9.1}$$

Let's do an example to see how this works. Assume there are two jobs, A and B. A, because it's our most precious job, is given a higher priority by assigning it a nice value of -5; B, because we hates it[3], just has the default priority (nice value equal to 0). This means $\texttt{weight}_A$ (from the table) is 3121, whereas $\texttt{weight}_B$ is 1024. If you then compute the time slice of each job, you'll find that A's time slice is about $\frac{3}{4}$ of `sched_latency` (hence, 36 ms), and B's about $\frac{1}{4}$ (hence, 12 ms).

In addition to generalizing the time slice calculation, the way CFS calculates `vruntime` must also be adapted. Here is the new formula, which takes the actual run time that process i has accrued ($\texttt{runtime}_i$) and scales it inversely by the weight of the process, by dividing the default weight of 1024 ($\texttt{weight}_0$) by its weight, $\texttt{weight}_i$. In our running example, A's `vruntime` will accumulate at one-third the rate of B's.

$$\texttt{vruntime}_i = \texttt{vruntime}_i + \frac{\texttt{weight}_0}{\texttt{weight}_i} \cdot \texttt{runtime}_i \tag{9.2}$$

One smart aspect of the construction of the table of weights above is that the table preserves CPU proportionality ratios when the difference in nice values is constant. For example, if process A instead had a nice value of 5 (not -5), and process B had a nice value of 10 (not 0), CFS would schedule them in exactly the same manner as before. Run through the math yourself to see why.

Using Red-Black Trees

One major focus of CFS is efficiency, as stated above. For a scheduler, there are many facets of efficiency, but one of them is as simple as this: when the scheduler has to find the next job to run, it should do so as quickly as possible. Simple data structures like lists don't scale: modern systems sometimes are comprised of 1000s of processes, and thus searching through a long-list every so many milliseconds is wasteful.

CFS addresses this by keeping processes in a **red-black tree** [B72]. A red-black tree is one of many types of balanced trees; in contrast to a simple binary tree (which can degenerate to list-like performance under worst-case insertion patterns), balanced trees do a little extra work to maintain low depths, and thus ensure that operations are logarithmic (and not linear) in time.

CFS does not keep *all* process in this structure; rather, only running (or runnable) processes are kept therein. If a process goes to sleep (say, waiting on an I/O to complete, or for a network packet to arrive), it is removed from the tree and kept track of elsewhere.

Let's look at an example to make this more clear. Assume there are ten jobs, and that they have the following values of `vruntime`: 1, 5, 9, 10, 14, 18, 17, 21, 22, and 24. If we kept these jobs in an ordered list, finding the next job to run would be simple: just remove the first element. However,

[3]Yes, yes, we are using bad grammar here on purpose, please don't send in a bug fix. Why? Well, just a most mild of references to the Lord of the Rings, and our favorite anti-hero Gollum, nothing to get too excited about.

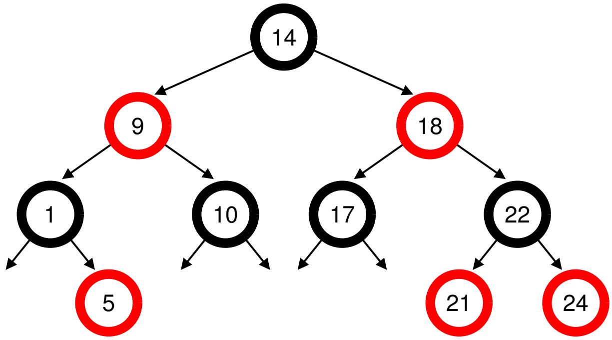


Figure 9.5: **CFS Red-Black Tree**

when placing that job back into the list (in order), we would have to scan the list, looking for the right spot to insert it, an $O(n)$ operation. Any search is also quite inefficient, also taking linear time on average.

Keeping the same values in a red-black tree makes most operations more efficient, as depicted in Figure 9.5. Processes are ordered in the tree by `vruntime`, and most operations (such as insertion and deletion) are logarithmic in time, i.e., $O(\log n)$. When n is in the thousands, logarithmic is noticeably more efficient than linear.

Dealing With I/O And Sleeping Processes

One problem with picking the lowest `vruntime` to run next arises with jobs that have gone to sleep for a long period of time. Imagine two processes, A and B, one of which (A) runs continuously, and the other (B) which has gone to sleep for a long period of time (say, 10 seconds). When B wakes up, its `vruntime` will be 10 seconds behind A's, and thus (if we're not careful), B will now monopolize the CPU for the next 10 seconds while it catches up, effectively starving A.

CFS handles this case by altering the `vruntime` of a job when it wakes up. Specifically, CFS sets the `vruntime` of that job to the minimum value found in the tree (remember, the tree only contains running jobs) [B+18]. In this way, CFS avoids starvation, but not without a cost: jobs that sleep for short periods of time frequently do not ever get their fair share of the CPU [AC97].

Other CFS Fun

CFS has many other features, too many to discuss at this point in the book. It includes numerous heuristics to improve cache performance, has strategies for handling multiple CPUs effectively (as discussed later in the book), can schedule across large groups of processes (instead of treating

TIP: USE EFFICIENT DATA STRUCTURES WHEN APPROPRIATE

In many cases, a list will do. In many cases, it will not. Knowing which data structure to use when is a hallmark of good engineering. In the case discussed herein, simple lists found in earlier schedulers simply do not work well on modern systems, particular in the heavily loaded servers found in datacenters. Such systems contain thousands of active processes; searching through a long list to find the next job to run on each core every few milliseconds would waste precious CPU cycles. A better structure was needed, and CFS provided one by adding an excellent implementation of a red-black tree. More generally, when picking a data structure for a system you are building, carefully consider its access patterns and its frequency of usage; by understanding these, you will be able to implement the right structure for the task at hand.

each process as an independent entity), and many other interesting features. Read recent research, starting with Bouron [B+18], to learn more.

9.8 Summary

We have introduced the concept of proportional-share scheduling and briefly discussed three approaches: lottery scheduling, stride scheduling, and the Completely Fair Scheduler (CFS) of Linux. Lottery uses randomness in a clever way to achieve proportional share; stride does so deterministically. CFS, the only "real" scheduler discussed in this chapter, is a bit like weighted round-robin with dynamic time slices, but built to scale and perform well under load; to our knowledge, it is the most widely used fair-share scheduler in existence today.

No scheduler is a panacea, and fair-share schedulers have their fair share of problems. One issue is that such approaches do not particularly mesh well with I/O [AC97]; as mentioned above, jobs that perform I/O occasionally may not get their fair share of CPU. Another issue is that they leave open the hard problem of ticket or priority assignment, i.e., how do you know how many tickets your browser should be allocated, or to what nice value to set your text editor? Other general-purpose schedulers (such as the MLFQ we discussed previously, and other similar Linux schedulers) handle these issues automatically and thus may be more easily deployed.

The good news is that there are many domains in which these problems are not the dominant concern, and proportional-share schedulers are used to great effect. For example, in a **virtualized** data center (or **cloud**), 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. The idea can also be extended to other resources; see Waldspurger [W02] for further details on how to proportionally share memory in VMWare's ESX Server.

References

[AC97] "Extending Proportional-Share Scheduling to a Network of Workstations" by 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.*

[B+18] "The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS" by J. Bouron, S. Chevalley, B. Lepers, W. Zwaenepoel, R. Gouicem, J. Lawall, G. Muller, J. Sopena. USENIX ATC '18, July 2018, Boston, Massachusetts. *A recent, detailed work comparing Linux CFS and the FreeBSD schedulers. An excellent overview of each scheduler is also provided. The result of the comparison: inconclusive (in some cases CFS was better, and in others, ULE (the BSD scheduler), was. Sometimes in life there are no easy answers.*

[B72] "Symmetric binary B-Trees: Data Structure And Maintenance Algorithms" by Rudolf Bayer. Acta Informatica, Volume 1, Number 4, December 1972. *A cool balanced tree introduced before you were born (most likely). One of many balanced trees out there; study your algorithms book for more alternatives!*

[D82] "Why Numbering Should Start At Zero" by Edsger Dijkstra, August 1982. Available: `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.*

[K+15] "Profiling A Warehouse-scale Computer" by S. Kanev, P. Ranganathan, J. P. Darago, K. Hazelwood, T. Moseley, G. Wei, D. Brooks. ISCA '15, June, 2015, Portland, Oregon. *A fascinating study of where the cycles go in modern data centers, which are increasingly where most of computing happens. Almost 20% of CPU time is spent in the operating system, 5% in the scheduler alone!*

[J09] "Inside The Linux 2.6 Completely Fair Scheduler" by M. Tim Jones. December 15, 2009. `http://ostep.org/Citations/inside-cfs.pdf`. *A simple overview of CFS from its earlier days. CFS was created by Ingo Molnar in a short burst of creativity which led to a 100K kernel patch developed in 62 hours.*

[KL88] "A Fair Share Scheduler" by 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" by 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" by 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" by 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 (Simulation)

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). Multi-threaded 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.

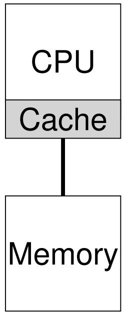


Figure 10.1: **Single CPU With Cache**

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.

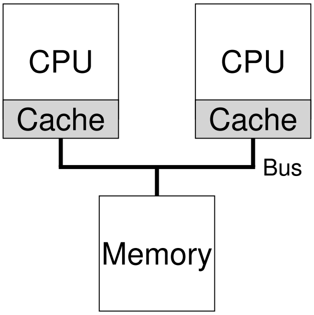


Figure 10.2: **Two CPUs With Caches Sharing Memory**

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. 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 in 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

```
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**

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 same head element, leading to all sorts of problems (such as an attempted double free of the head element at Line 10, 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** [TTG95]. 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.

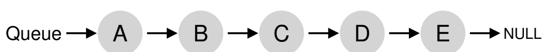
10.4 Single-Queue Scheduling

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).

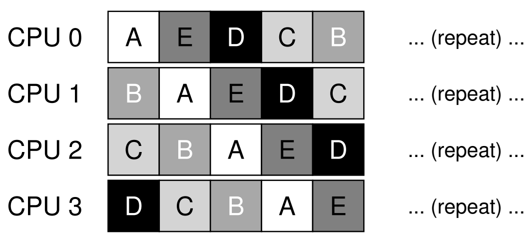
However, SQMS has obvious shortcomings. The first problem is a lack of **scalability**. To ensure the scheduler works correctly on multiple CPUs, the developers will have inserted some form of **locking** into the code, as described above. Locks ensure that when SQMS code accesses the single queue (say,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

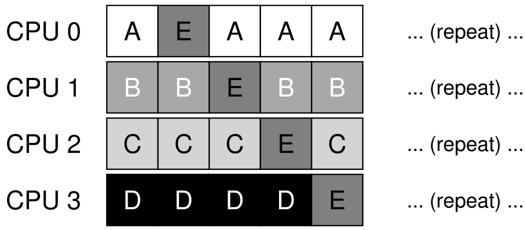


Over time, assuming each job runs for a time slice and then another job is chosen, here is a possible job schedule across CPUs:



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.

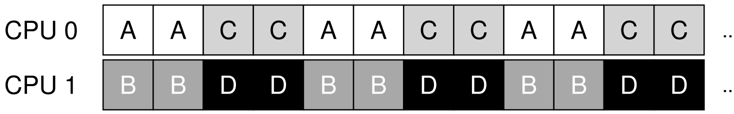
10.5 Multi-Queue Scheduling

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:

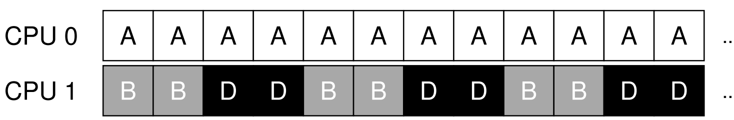
Depending on the queue scheduling policy, each CPU now has two jobs to choose from when deciding what should run. For example, with **round robin**, the system might produce a schedule that looks like this:



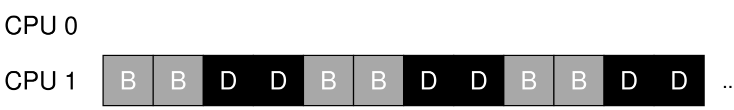
MQMS has a distinct advantage of SQMS in that it should be inherently more scalable. As the number of CPUs grows, so too does the number of queues, and thus lock and cache contention should not become a central problem. In addition, MQMS intrinsically provides cache affinity; jobs stay on the same CPU 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 two scheduling queues, and resulting timeline, will look like this:



How terrible – CPU 0 is idle! *(insert dramatic and sinister music here)* And thus our CPU usage timeline looks quite 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?

> 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?

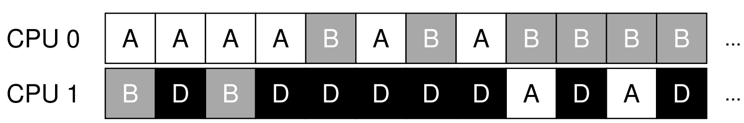
The obvious answer to this query is to move jobs around, a technique which we (once again) refer to as **migration**. By migrating a job from one CPU to another, true load balance can be achieved.

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?

[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.

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" by 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" by 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" by 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" by 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" by 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" by 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" by 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" by 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.*

[TTG95] "Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors" by Josep Torrellas, Andrew Tucker, Anoop Gupta. Journal of Parallel and Distributed Computing, Volume 24:2, February 1995. *This is not the first paper on the topic, but it has citations to earlier work, and is a more readable and practical paper than some of the earlier queuing-based analysis papers.*

Homework (Simulation)

In this homework, we'll use `multi.py` to simulate a multi-processor CPU scheduler, and learn about some of its details. Read the related README for more information about the simulator and its options.

Questions

1. To start things off, let's learn how to use the simulator to study how to build an effective multi-processor scheduler. The first simulation will run just one job, which has a run-time of 30, and a working-set size of 200. Run this job (called job 'a' here) on one simulated CPU as follows: `./multi.py -n 1 -L a:30:200`. How long will it take to complete? Turn on the `-c` flag to see a final answer, and the `-t` flag to see a tick-by-tick trace of the job and how it is scheduled.

2. Now increase the cache size so as to make the job's working set (size=200) fit into the cache (which, by default, is size=100); for example, run `./multi.py -n 1 -L a:30:200 -M 300`. Can you predict how fast the job will run once it fits in cache? (hint: remember the key parameter of the warm_rate, which is set by the -r flag) Check your answer by running with the solve flag (`-c`) enabled.

3. One cool thing about `multi.py` is that you can see more detail about what is going on with different tracing flags. Run the same simulation as above, but this time with time_left tracing enabled (`-T`). This flag shows both the job that was scheduled on a CPU at each time step, as well as how much run-time that job has left after each tick has run. What do you notice about how that second column decreases?

4. Now add one more bit of tracing, to show the status of each CPU cache for each job, with the `-C` flag. For each job, each cache will either show a blank space (if the cache is cold for that job) or a 'w' (if the cache is warm for that job). At what point does the cache become warm for job 'a' in this simple example? What happens as you change the warmup_time parameter (`-w`) to lower or higher values than the default?

5. At this point, you should have a good idea of how the simulator works for a single job running on a single CPU. But hey, isn't this a multi-processor CPU scheduling chapter? Oh yeah! So let's start working with multiple jobs. Specifically, let's run the following three jobs on a two-CPU system (i.e., type `./multi.py -n 2 -L a:100:100,b:100:50,c:100:50`) Can you predict how long this will take, given a round-robin centralized scheduler? Use `-c` to see if you were right, and then dive down into details with `-t`

to see a step-by-step and then `-C` to see whether caches got warmed effectively for these jobs. What do you notice?

6. Now we'll apply some explicit controls to study **cache affinity**, as described in the chapter. To do this, you'll need the `-A` flag. This flag can be used to limit which CPUs the scheduler can place a particular job upon. In this case, let's use it to place jobs 'b' and 'c' on CPU 1, while restricting 'a' to CPU 0. This magic is accomplished by typing this `./multi.py -n 2 -L a:100:100,b:100:50, c:100:50 -A a:0,b:1,c:1`; don't forget to turn on various tracing options to see what is really happening! Can you predict how fast this version will run? Why does it do better? Will other combinations of 'a', 'b', and 'c' onto the two processors run faster or slower?

7. One interesting aspect of caching multiprocessors is the opportunity for better-than-expected speed up of jobs when using multiple CPUs (and their caches) as compared to running jobs on a single processor. Specifically, when you run on N CPUs, sometimes you can speed up by more than a factor of N, a situation entitled **super-linear speedup**. To experiment with this, use the job description here (`-L a:100:100,b:100:100,c:100:100`) with a small cache (`-M 50`) to create three jobs. Run this on systems with 1, 2, and 3 CPUs (`-n 1`, `-n 2`, `-n 3`). Now, do the same, but with a larger per-CPU cache of size 100. What do you notice about performance as the number of CPUs scales? Use `-c` to confirm your guesses, and other tracing flags to dive even deeper.

8. One other aspect of the simulator worth studying is the per-CPU scheduling option, the `-p` flag. Run with two CPUs again, and this three job configuration (`-L a:100:100,b:100:50,c:100:50`). How does this option do, as opposed to the hand-controlled affinity limits you put in place above? How does performance change as you alter the 'peek interval' (`-P`) to lower or higher values? How does this per-CPU approach work as the number of CPUs scales?

9. Finally, feel free to just generate random workloads and see if you can predict their performance on different numbers of processors, cache sizes, and scheduling options. If you do this, you'll soon be a **multi-processor scheduling master**, which is a pretty awesome thing to be. Good luck!

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 (page 2).

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?

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 ineffec-

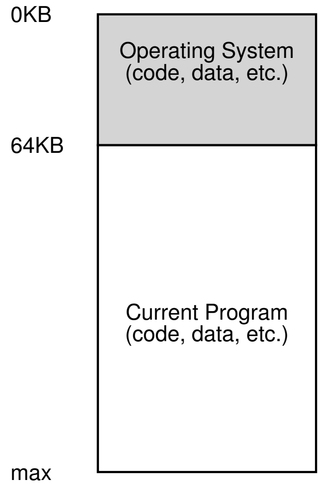


Figure 13.1: **Operating Systems: The Early Days**

tive) 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), 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 (as shown in Figure 13.2, page 3).

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.

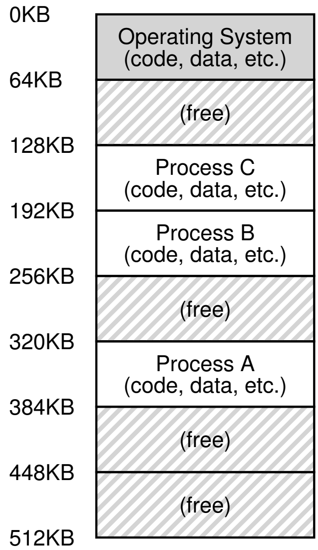


Figure 13.2: **Three Processes: Sharing Memory**

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 4), we have a tiny address space (only 16KB)[1]. The program code lives at the top of the address space

[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.

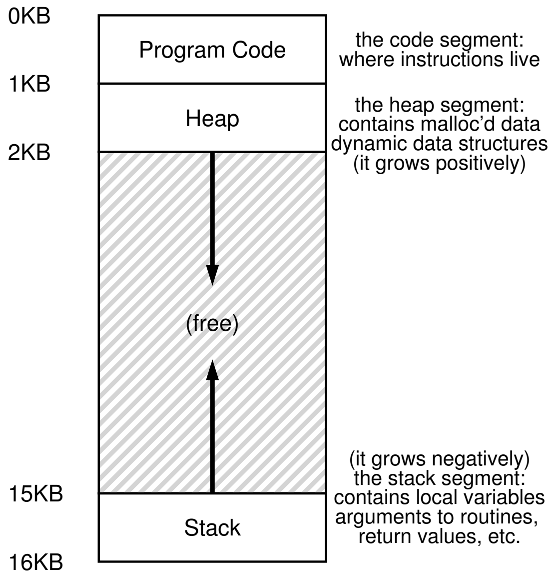


Figure 13.3: **An Example Address Space**

(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.

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 pro-

[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.

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.

cesses. 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.

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].

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!

[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!

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 (`va.c`) 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:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
  printf("location of code : %p\n", main);
  printf("location of heap : %p\n", malloc(100e6));
  int x = 3;
  printf("location of stack: %p\n", &x);
  return x;
}
```

When run on a 64-bit Mac, 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.

References

[BH70] "The Nucleus of a Multiprogramming System" by 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" by F. J. Corbato, 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" by Jack B. Dennis, 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" by 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" by J. McCarthy. Management and the Computer of the Future, MIT Press, Cambridge, MA, 1962. *Probably McCarthy's earliest recorded paper on time sharing. 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" by J. McCarthy, S. Boilen, E. Fredkin, J. C. R. Licklider. AFIPS '63 (Spring), New York, NY, May 1963. *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" by John McCarthy. 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.*

[NS07] "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation" by N. Nethercote, J. Seward. PLDI 2007, San Diego, California, June 2007. *Valgrind is a lifesaver of a program for those who use unsafe languages like C. Read this paper to learn about its very cool binary instrumentation techniques – it's really quite impressive.*

[R+89] "Mach: A System Software kernel" by R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, M. 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" by 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" by M. M. Swift, B. N. Bershad, H. M. Levy. SOSP '03. *The first paper to show how microkernel-like thinking can improve operating system reliability.*

Homework (Code)

In this homework, we'll just learn about a few useful tools to examine virtual memory usage on Linux-based systems. This will only be a brief hint at what is possible; you'll have to dive deeper on your own to truly become an expert (as always!).

Questions

1. The first Linux tool you should check out is the very simple tool `free`. First, type `man free` and read its entire manual page; it's short, don't worry!
2. Now, run `free`, perhaps using some of the arguments that might be useful (e.g., `-m`, to display memory totals in megabytes). How much memory is in your system? How much is free? Do these numbers match your intuition?
3. Next, create a little program that uses a certain amount of memory, called `memory-user.c`. This program should take one command-line argument: the number of megabytes of memory it will use. When run, it should allocate an array, and constantly stream through the array, touching each entry. The program should do this indefinitely, or, perhaps, for a certain amount of time also specified at the command line.
4. Now, while running your `memory-user` program, also (in a different terminal window, but on the same machine) run the `free` tool. How do the memory usage totals change when your program is running? How about when you kill the `memory-user` program? Do the numbers match your expectations? Try this for different amounts of memory usage. What happens when you use really large amounts of memory?
5. Let's try one more tool, known as `pmap`. Spend some time, and read the `pmap` manual page in detail.
6. To use `pmap`, you have to know the **process ID** of the process you're interested in. Thus, first run `ps auxw` to see a list of all processes; then, pick an interesting one, such as a browser. You can also use your `memory-user` program in this case (indeed, you can even have that program call `getpid()` and print out its PID for your convenience).
7. Now run `pmap` on some of these processes, using various flags (like `-X`) to reveal many details about the process. What do you see? How many different entities make up a modern address space, as opposed to our simple conception of code/stack/heap?
8. Finally, let's run `pmap` on your `memory-user` program, with different amounts of used memory. What do you see here? Does the output from `pmap` match your expectations?

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.

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

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 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 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

[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!

utilized. For example, to allocate space for a double-precision floating point value, you simply do this:

```
double *d = (double *) malloc(sizeof(double));
```

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 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
```

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.

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.**

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.

[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?

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.

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...

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.

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" by R. Hastings, B. Joyce. USENIX Winter '92. *The paper behind the cool Purify tool, now a commercial product.*

[KR88] "The C Programming Language" by Brian Kernighan, 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" by G. Novark, E. D. Berger, B. G. Zorn. PLDI 2007, San Diego, California. *A cool paper on finding and correcting memory errors automatically, and a great overview of many common errors in C and C++ programs. An extended version of this paper is available CACM (Volume 51, Issue 12, December 2008).*

[SN05] "Using Valgrind to Detect Undefined Value Errors with Bit-precision" by J. Seward, N. Nethercote. USENIX '05. *How to use valgrind to find certain types of errors.*

[SR05] "Advanced Programming in the UNIX Environment" by W. Richard Stevens, 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" by T. 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 tools are the debugger (e.g., `gdb`) and a memory-bug detector called `valgrind` [SN05].

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 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 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:

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 interface of the client, thus requiring no changes to said client.

```
void func() {
    int x = 3000; // thanks, Perry.
    x = x + 3;    // 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 a Mac 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), observe 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)

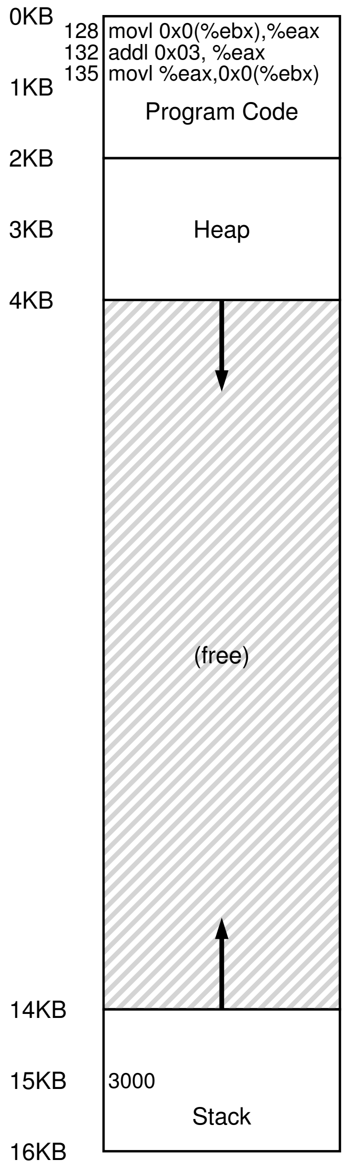


Figure 15.1: **A Process And Its Address Space**

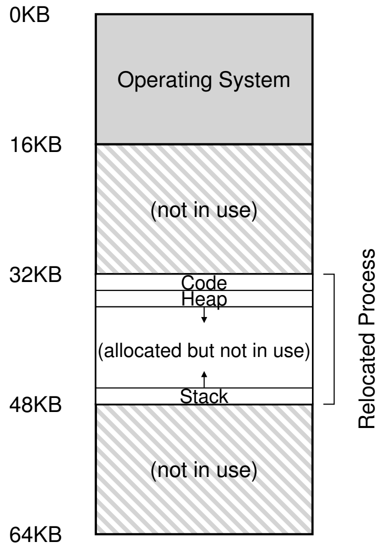


Figure 15.2: **Physical Memory with a Single Relocated Process**

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?

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

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].

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
```

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

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.

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].

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.

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

[1] Is there anything other than "havoc" that can be "wreaked"? [W17]

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.

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**

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.

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.

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**

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 between 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

OS @ boot (kernel mode)	Hardware	(No Program Yet)
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		

Figure 15.5: **Limited Direct Execution (Dynamic Relocation) @ Boot**

execute instructions that it shouldn't. Bye bye, misbehaving process; it's been nice knowing you.

Figures 15.5 and 15.6 (page 12) illustrate much of the hardware/OS interaction in a timeline. The first figure shows what the OS does at boot time to ready the machine for use, and the second shows 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 (middle of second figure), 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 figures, 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.

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

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
To start process A: allocate entry in process table alloc memory for process set base/bound 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 perform fetch	
		Execute instruction
	if explicit load/store: ensure address is legal translate virtual address perform load/store	
		(A runs...)
	Timer interrupt move to **kernel mode** jump to handler	
Handle timer decide: stop A, run B 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 kill process B deallocate B's memory free B's entry in process table		

Figure 15.6: **Limited Direct Execution (Dynamic Relocation) @ Runtime**

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" by W.C. McGee. IBM Systems Journal, Volume 4:3, 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" by Kenneth D. A. Pillay. Microprocessors & Microsystems archive, Volume 14:7 (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" by 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" by Mark Smotherman. May 2004. `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" by 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.*

[W17] Answer to footnote: "Is there anything other than havoc that can be wreaked?" by Waciuma Wanjohi. October 2017. *Amazingly, this enterprising reader found the answer via google's Ngram viewing tool (available at the following URL: `http://books.google.com/ngrams`). The answer, thanks to Mr. Wanjohi: "It's only since about 1970 that 'wreak havoc' has been more popular than 'wreak vengeance'. In the 1800s, the word wreak was almost always followed by 'his/their vengeance'." Apparently, when you wreak, you are up to no good, but at least wreakers have some options now.*

Homework (Simulation)

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

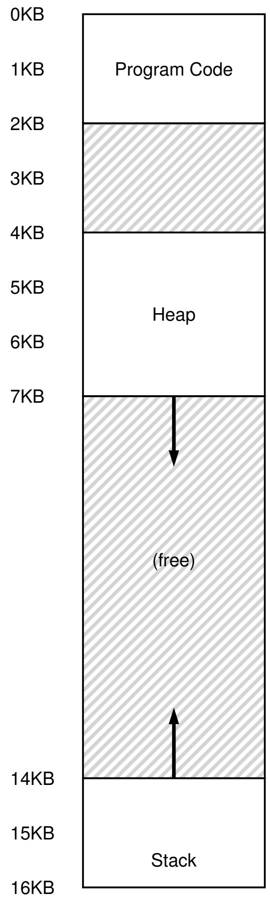


Figure 16.1: **An Address Space (Again)**

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).

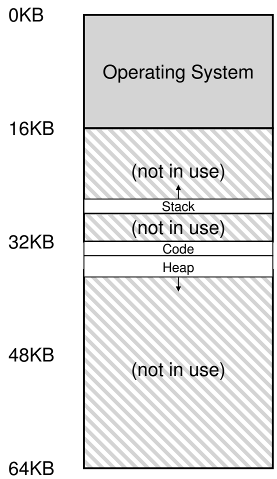


Figure 16.2: **Placing Segments In Physical Memory**

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	3K
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 has a size of 3KB. The size segment here is exactly the same as the bounds register introduced previously; it tells the hardware exactly how many bytes are valid in this segment (and thus, enables the hardware to determine when a program has made an illegal access outside of those bounds).

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, as you can see visually in Figure 16.1, page 2). When the refer-

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 out why your code keeps faulting.

ence 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.

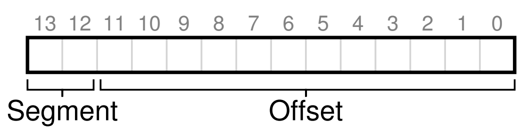
Now let's look at an address in the heap, virtual address 4200 (again refer to Figure 16.1). If we just add the virtual address 4200 to the base of the heap (34KB), we get a physical address of 39016, which is *not* the correct physical address. What we need to first do is extract the *offset* into the 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 (i.e., a virtual address of 7KB or greater), 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**.

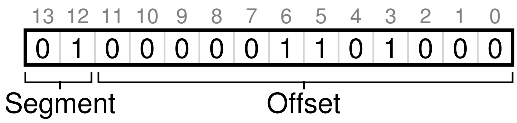
16.2 Which Segment Are We Referring To?

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:

```
// get top 2 bits of 14-bit VA
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
// now get offset
Offset  = VirtualAddress & OFFSET_MASK
if (Offset >= Bounds[Segment])
    RaiseException(PROTECTION_FAULT)
else
    PhysAddr = Base[Segment] + Offset
    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. To fully utilize the virtual address space (and avoid an unused segment), some systems put code in the same segment as the heap and thus use only one bit to select which segment to use [LL82].

Another issue with using the top so many bits to select a segment is that it limits use of the virtual address space. Specifically, each segment is limited to a *maximum size*, which in our example is 4KB (using the top two bits to choose segments implies the 16KB address space gets chopped into four pieces, or 4KB in this example). If a running program wishes to grow a segment (say the heap, or the stack) beyond that maximum, the program is out of luck.

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* (i.e., towards lower addresses). In physical memory, it "starts" at 28KB[1] 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 (max 4K)	Grows Positive?
Code_{00}	32K	2K	1
Heap_{01}	34K	3K	1
Stack_{11}	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 or equal to the segment's current size (in this case, 2KB).

[1] Although we say, for simplicity, that the stack "starts" at 28KB, this value is actually the byte just *below* the location of the backward growing region; the first valid byte is actually 28KB minus 1. In contrast, forward-growing regions start at the address of the first byte of the segment. We take this approach because it makes the math to compute the physical address straightforward: the physical address is just the base plus the negative offset.

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 (max 4K)	Grows Positive?	Protection
$Code_{00}$	32K	2K	1	Read-Execute
$Heap_{01}$	34K	3K	1	Read-Write
$Stack_{11}$	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

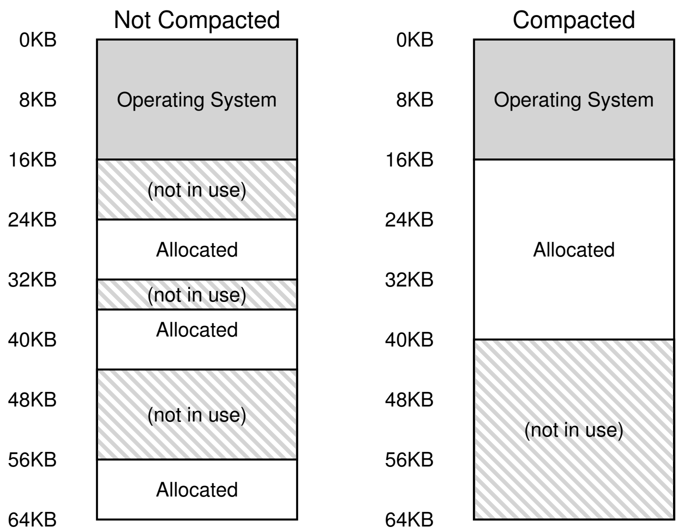


Figure 16.6: **Non-compacted and Compacted Memory**

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 and support a large and sparse virtual address space per process.

However, segmentation raises a number of new issues for the operating system. 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 is OS interaction when segments grow (or perhaps shrink). For example, a program may call `malloc()` to allocate an object. In some cases, the existing heap will be able to service the request, and thus

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.

`malloc()` will find free space for the object and return a pointer to it to the caller. In others, however, the heap segment itself may need to grow. In this case, the memory-allocation library will perform a system call to grow the heap (e.g., the traditional UNIX `sbrk()` system call). The OS will then (usually) provide more space, updating the segment size register to the new (bigger) size, and informing the library of success; the library can then allocate space for the new object and return successfully to the calling program. Do note that the OS could reject the request, if no more physical memory is available, or if it decides that the calling process already has too much.

The last, and perhaps most 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.

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. Similar problems could occur when a request to grow a segment arrives; if the next so many bytes of physical space are not available, the OS will have to reject the request, even though there may be free bytes available elsewhere in physical memory.

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. Compaction also (ironically) makes requests to grow existing segments hard to serve, and may thus cause further rearrangement to accommodate such requests.

A simpler approach might instead be 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 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.

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" by F. J. Corbato, 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" by Robert C. Daley and Jack B. Dennis. Communications of the ACM, Volume 11:5, 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" by 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" by A. W. Holt. Communications of the ACM, Volume 4:10, 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" by 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" by 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" by 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" by Henry M. Levy, Peter H. Lipman. IEEE Computer, Volume 15:3, 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" by B. Randell and C.J. Kuehner. Communications of the ACM, Volume 11:5, 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" by Brian Randell. Communications of the ACM, Volume 12:7, July 1969. *One of the earliest papers to discuss fragmentation.*

[W+95] "Dynamic Storage Allocation: A Survey and Critical Review" by Paul R. Wilson, Mark S. Johnstone, Michael Neely, David Boles. International Workshop on Memory Management, Scotland, UK, September 1995. *A great survey paper on memory allocators.*

Homework (Simulation)

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. Assume we want to generate a problem where roughly 90% of the randomly-generated virtual addresses are valid (not segmentation violations). How should you configure the simulator to do so? Which parameters are important to getting this outcome?

5. Can you run the simulator such that no virtual addresses are valid? How?

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(!).

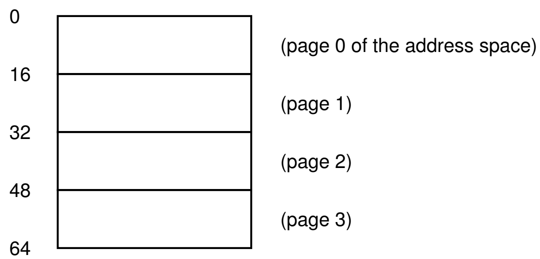


Figure 18.1: **A Simple 64-byte Address Space**

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.

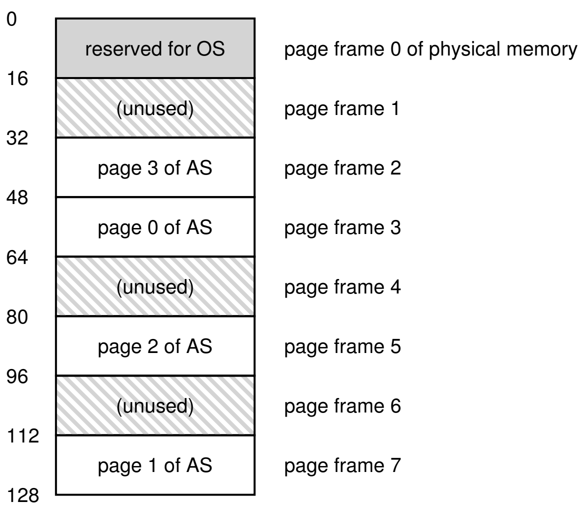


Figure 18.2: **A 64-Byte Address Space In A 128-Byte Physical Memory**

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 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:

Va5	Va4	Va3	Va2	Va1	Va0

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:

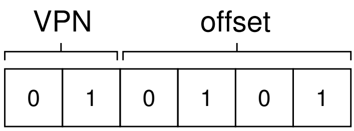
VPN		offset			
Va5	Va4	Va3	Va2	Va1	Va0

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).

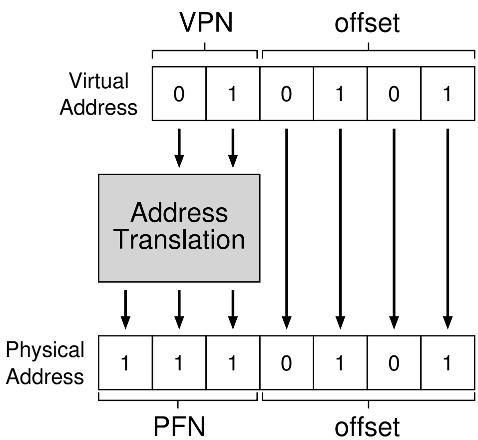


Figure 18.3: **The Address Translation Process**

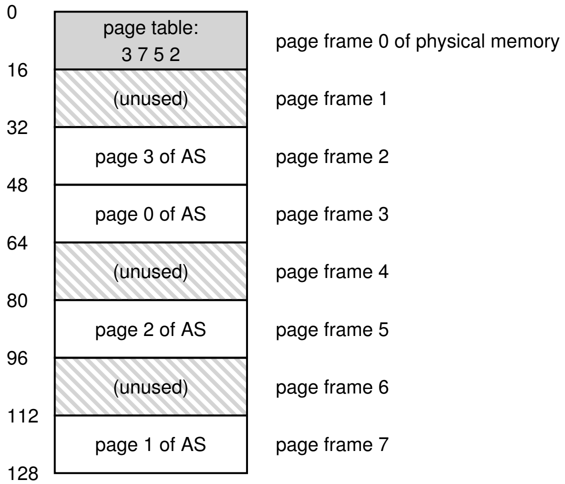


Figure 18.4: **Example: Page Table in Kernel Physical Memory**

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!

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

> 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).

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 (page 5) is a picture of a page table in OS memory; see the tiny set of translations in there?

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.

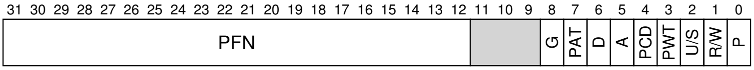


Figure 18.5: **An x86 Page Table Entry (PTE)**

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.

ASIDE: WHY NO VALID BIT?

You may notice that in the Intel example, there are no separate valid and present bits, but rather just a present bit (P). If that bit is set (P=1), it means the page is both present and valid. If not (P=0), it means that the page may not be present in memory (but is valid), or may not be valid. An access to a page with P=0 will trigger a trap to the OS; the OS must then use additional structures it keeps to determine whether the page is valid (and thus perhaps should be swapped back in) or not (and thus the program is attempting to access memory illegally). This sort of judiciousness is common in hardware, which often just provide the minimal set of features upon which the OS can build a full service.

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 bitwise OR'd with the offset to form the final address as follows:

```
offset   = VirtualAddress & OFFSET_MASK
PhysAddr = (PFN << SHIFT) | offset
```

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 (page 9) shows the 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

```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT

// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))

// Fetch the PTE
PTE = AccessMemory(PTEAddr)

// Check if process can access the page
if (PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False)
    RaiseException(PROTECTION_FAULT)
else
    // Access is OK: form physical address and fetch it
    offset   = VirtualAddress & OFFSET_MASK
    PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
    Register = AccessMemory(PhysAddr)
```

Figure 18.6: **Accessing Memory With Paging**

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:

```
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.

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).

[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.

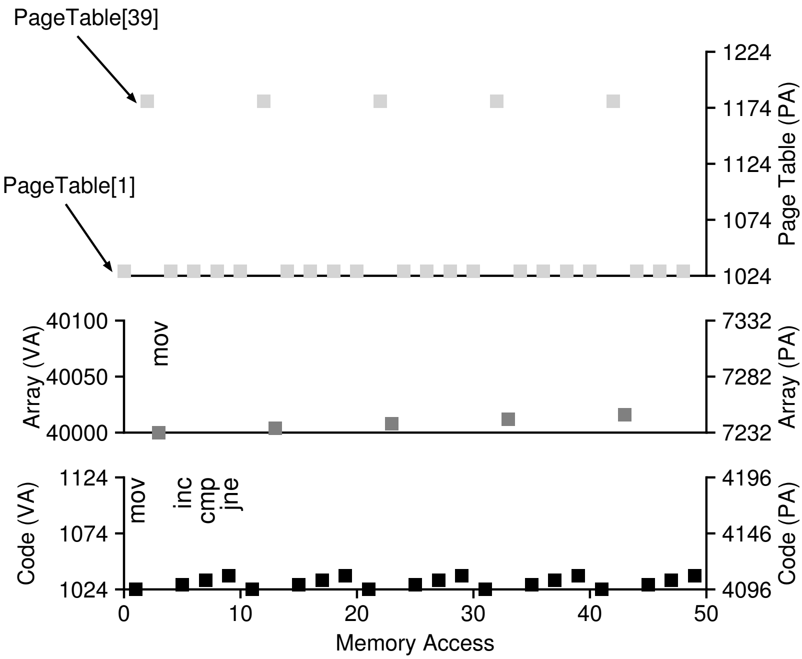


Figure 18.7: **A Virtual (And Physical) Memory Trace**

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 11). 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" by T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner. IRE Trans. EC-11, 2, 1962. 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" by S. H. Lavington. Communications of the ACM, Volume 21:1, January 1978. *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 (Simulation)

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, run with these flags:

   ```
   -P 1k -a 1m -p 512m -v -n 0
   -P 1k -a 2m -p 512m -v -n 0
   -P 1k -a 4m -p 512m -v -n 0
   ```

   Then, to understand how linear page table size changes as page size grows:

   ```
   -P 1k -a 1m -p 512m -v -n 0
   -P 2k -a 1m -p 512m -v -n 0
   -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 not use 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:

   ```
   -P 1k -a 16k -p 32k -v -u 0
   -P 1k -a 16k -p 32k -v -u 25
   -P 1k -a 16k -p 32k -v -u 50
   -P 1k -a 16k -p 32k -v -u 75
   -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:

```
-P 8  -a 32   -p 1024 -v -s 1
-P 8k -a 32k  -p 1m   -v -s 2
-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** [CG68, 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].

```
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**

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).

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.

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 (page 4) 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

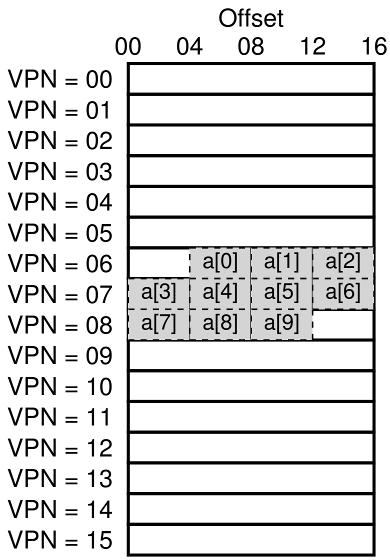


Figure 19.2: **Example: An Array In A Tiny Address Space**

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

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.

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.

```
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)**

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].

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 in-

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.

struction 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

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.

table, without necessitating hardware change. Another advantage is *simplicity*, as seen 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 do much on a miss: just raise an exception and let the OS TLB miss handler do the rest.

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.

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.

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

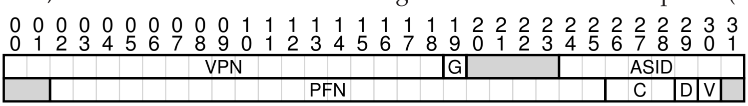


Figure 19.4: **A MIPS TLB Entry**

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, particularly 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**.

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).

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" by 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" by John Cocke, 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" by 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" by John F. Couleur, 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" by R.P. Case, 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". by Joe Heinrich. Prentice-Hall, June 1993. Available: http://cag.csail.mit.edu/raw/ . documents/R4400_Uman_book_Ed2.pdf *A manual, one that is surprisingly readable. Or is it?*

[HP06] "Computer Architecture: A Quantitative Approach" by 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" by 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" by 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" by Rafael H. Saavedra-Barrera. EECS Department, University of California, Berkeley. Technical Report No. UCB/CSD-92-684, February 1992.. *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" by 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" by David L. Weaver and Tom Germond. SPARC International, San Jose, California, September 2000. Available: `www.sparc.org/standards/SPARCV9.pdf`. *Another manual. I bet you were hoping for a more fun citation to end this chapter.*

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.

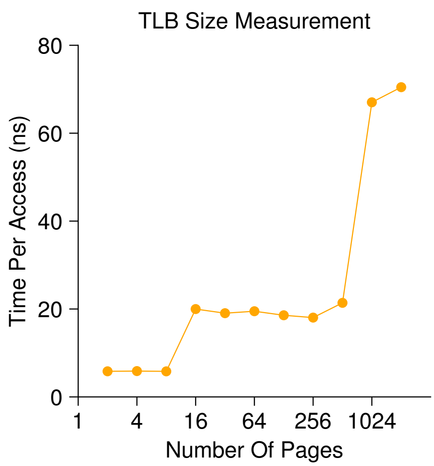


Figure 19.5: **Discovering TLB Sizes and Miss Costs**

Figure 19.5 (page 15) 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!

Questions

1. For timing, you'll need to use a timer (e.g., `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 (bash?) 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` or even `zplot`. Visualization usually makes the data much easier to digest; why do you think that is?
5. One thing to watch out for is compiler optimization. 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.

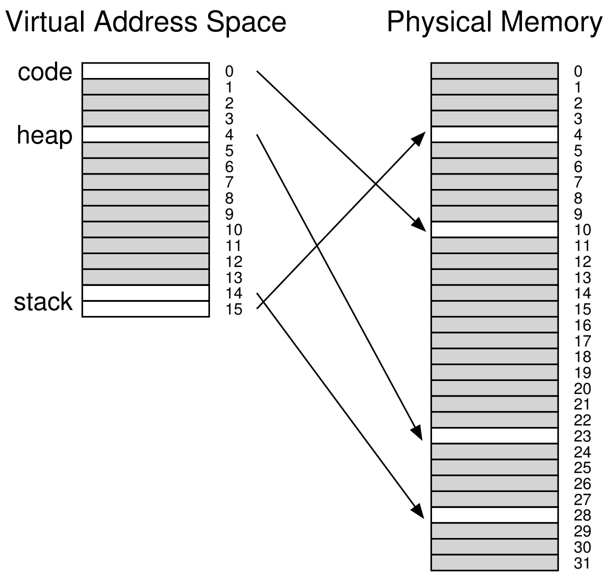


Figure 20.1: **A 16KB Address Space With 1KB Pages**

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	—	-	-
28	1	rw-	1	1
4	1	rw-	1	1

Figure 20.2: **A Page Table For 16KB Address Space**

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

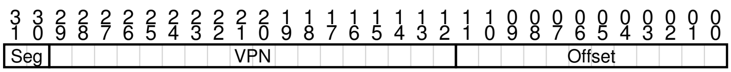
15) are mapped to physical pages 28 and 4, respectively. As you can see from the picture, *most* of the page table is unused, full of **invalid** entries. What a waste! And this is for a tiny 16KB address space. Imagine the page table of a 32-bit address space and all the potential wasted space in there! Actually, don't imagine such a thing; it's far too gruesome.

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.

> **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.

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 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.

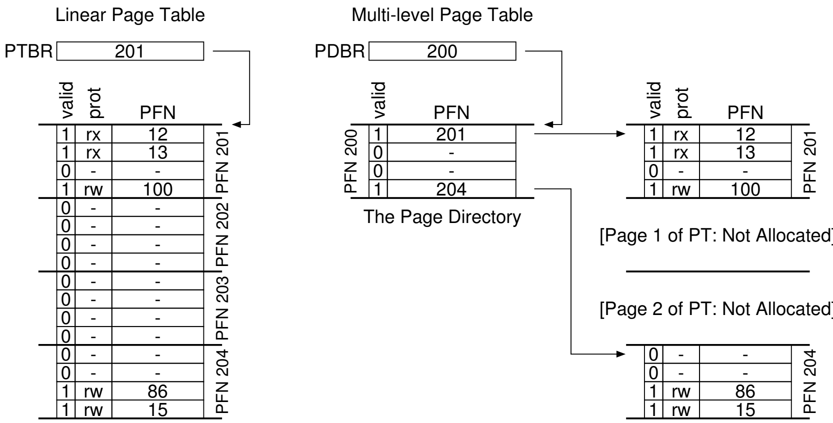


Figure 20.3: **Linear (Left) And Multi-Level (Right) Page Tables**

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 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 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 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

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.

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 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 (page 8) presents one example of such an address space.

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

[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.

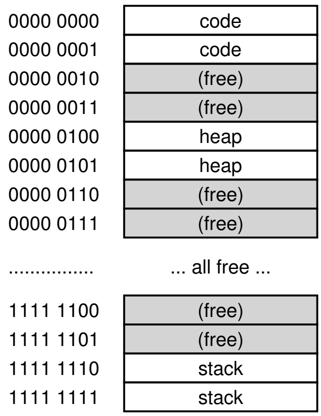
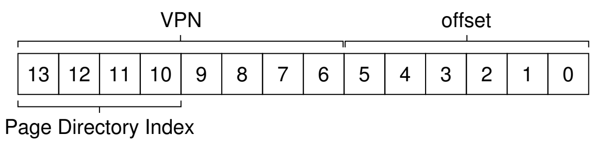


Figure 20.4: **A 16KB Address Space With 64-byte Pages**

in size. Thus, our page table is 1KB (256 × 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.

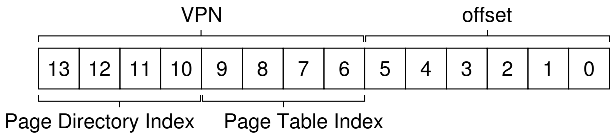
What we need to understand now is how to take a VPN and use it to index first into the page directory and then into the page of the page table. Remember that each is an array of entries; thus, all we need to figure out is how to construct the index for each from pieces of the VPN.

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.

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

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**

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."

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!

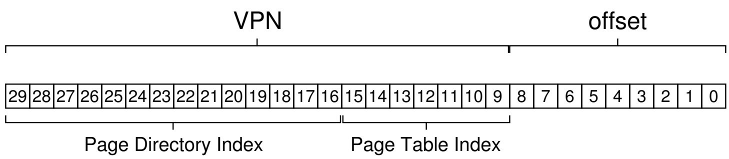
More Than Two Levels

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.

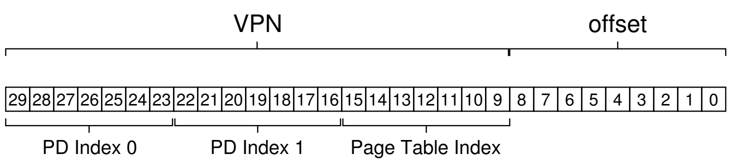
Remember our goal in constructing a multi-level page table: to make each piece of the page table fit within a single page. Thus far, we've only considered the page table itself; however, what if the page directory gets too big?

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

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)    // TLB Hit
  if (CanAccess(TlbEntry.ProtectBits) == True)
    Offset   = VirtualAddress & OFFSET_MASK
    PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
    Register = AccessMemory(PhysAddr)
  else
    RaiseException(PROTECTION_FAULT)
else                  // TLB Miss
  // first, get page directory entry
  PDIndex = (VPN & PD_MASK) >> PD_SHIFT
  PDEAddr = PDBR + (PDIndex * sizeof(PDE))
  PDE     = AccessMemory(PDEAddr)
  if (PDE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
  else
    // PDE is valid: now fetch PTE from page table
    PTIndex = (VPN & PT_MASK) >> PT_SHIFT
    PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
    PTE     = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
      RaiseException(SEGMENTATION_FAULT)
    else if (CanAccess(PTE.ProtectBits) == False)
      RaiseException(PROTECTION_FAULT)
    else
      TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
      RetryInstruction()
```

Figure 20.6: **Multi-level Page Table Control Flow**

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 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 up 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" by 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" by Bruce Jacob, 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" by Hank Levy, 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" by Mars Candy Corporation. Published at stores near you. *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 hate each other's guts, as any two chocolate barons should.*

[N+02] "Practical, Transparent Operating System Support for Superpages" by 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 (Simulation)

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,

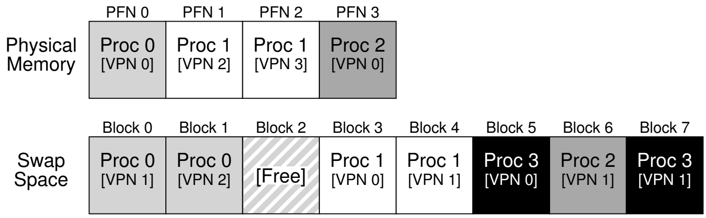


Figure 21.1: **Physical Memory and Swap Space**

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.

```
PFN = FindFreePhysicalPage()
if (PFN == -1)              // no free page found
    PFN = EvictPage()       // run replacement algorithm
DiskRead(PTE.DiskAddr, PFN) // sleep (waiting for I/O)
PTE.present = True          // update page table with present
PTE.PFN     = PFN           // bit and translation (PFN)
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" by F. Corbato, R. Steinberg. `www.takeourword.com/TOW146` (Page 4). *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" by Peter Denning. 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" by 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" by Hank Levy, P. Lipman. IEEE Computer, Vol. 15, No. 3, March 1982. *Not the first place where page clustering was used, but a clear and simple explanation of how such a mechanism works. We sure cite this paper a lot!*

Homework (Measurement)

This homework introduces you to a new tool, **vmstat**, and how it can be used to understand memory, CPU, and I/O usage. 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:

$$AMAT \;=\; T_M + (P_{Miss} \cdot T_D) \qquad (22.1)$$

where T_M represents the cost of accessing memory, T_D the cost of accessing disk, and P_{Miss} the probability of not finding the data in the cache (a miss); P_{Miss} varies from 0.0 to 1.0, and sometimes we refer to a percent miss rate instead of a probability (e.g., a 10% miss rate means $P_{Miss} = 0.10$). Note you always pay the cost of accessing the data in memory; when you miss, however, you must additionally pay the cost of fetching the data from disk.

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%, as 9 out of 10 references are in memory. The **miss rate** is thus 10% ($P_{Miss} = 0.1$). In general, $P_{Hit} + P_{Miss} = 1.0$; hit rate plus miss rate sum to 100%.

To calculate AMAT, we 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: $100ns + 0.1 \cdot 10ms$, which is $100ns + 1ms$, or 1.0001 ms, or about 1 millisecond. If our hit rate had instead been 99.9% ($P_{miss} = 0.001$), 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

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].

pages 0, 1, and 3 in the cache. The next three references are hits, but then 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

[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**

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 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 1.

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.

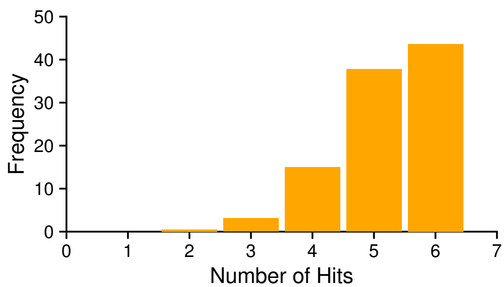


Figure 22.4: **Random Performance Over 10,000 Trials**

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 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.

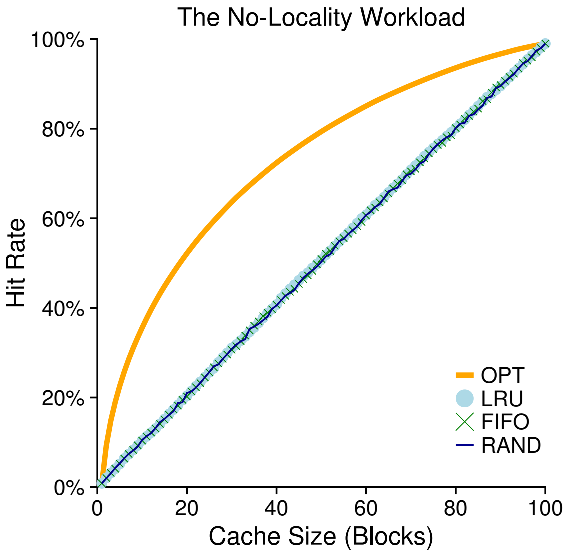


Figure 22.6: **The No-Locality Workload**

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.

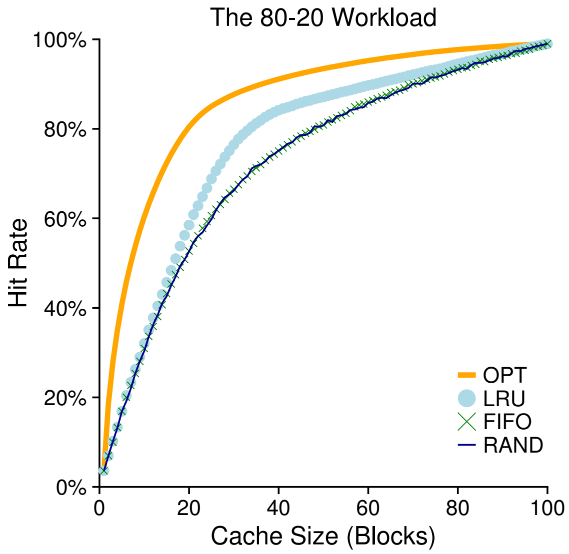


Figure 22.7: **The 80-20 Workload**

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.

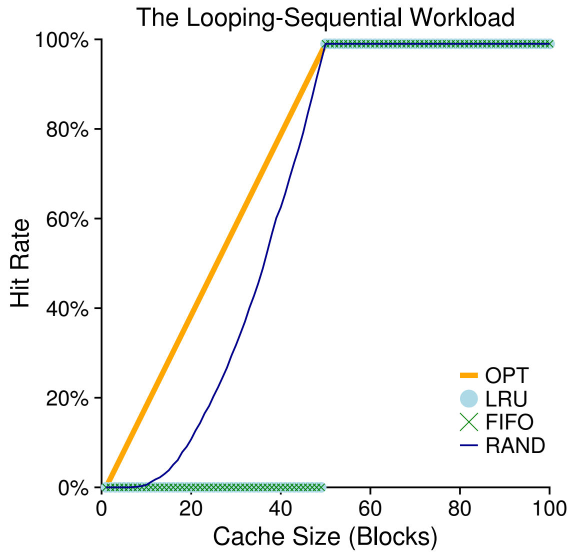


Figure 22.8: **The Looping Workload**

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 is 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.

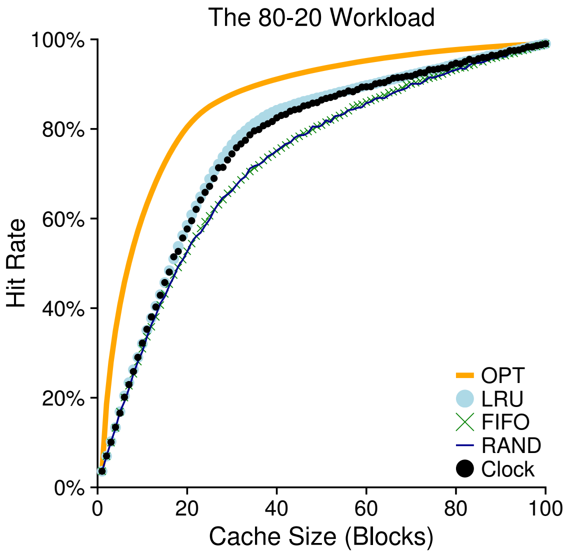


Figure 22.9: **The 80-20 Workload With Clock**

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.

For many years, the importance of replacement algorithms had decreased, as the discrepancy between memory-access and disk-access times was so large. Specifically, because paging to disk was so expensive, the cost of frequent paging was prohibitive; simply put, no matter how good your replacement algorithm was, if you were performing frequent replacements, your system became unbearably slow. Thus, the best solution was a simple (if intellectually unsatisfying) one: buy more memory.

However, recent innovations in much faster storage devices (e.g., Flash-based SSDs and Intel Optane products) have changed these performance ratios yet again, leading to a renaissance in page replacement algorithms. See [SS10,W+21] for recent work in this space.

References

[AD03] "Run-Time Adaptation in River" by Remzi H. Arpaci-Dusseau. ACM TOCS, 21:1, February 2003. *A summary of one of the authors' dissertation work on a system named River, 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" by 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" by L. A. Belady, R. A. Nelson, 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" by Hong-Tai Chou, 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" by 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" by 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" by Malcolm C. Easton, Ronald Fagin. Communications of the ACM, 21:10, October 1978. *A good discussion of cold- vs. warm-start misses.*

[FP89] "Electrochemically Induced Nuclear Fusion of Deuterium" by Martin Fleischmann, 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. Unfortunately, the results published (and widely publicized) by Pons and Fleischmann were 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, thus avoiding association with one of the biggest scientific goofs of the 20th century.*

[HP06] "Computer Architecture: A Quantitative Approach" by John Hennessy and David Patterson. Morgan-Kaufmann, 2006. *A marvelous book about computer architecture. Read it!*

[H87] "Aspects of Cache Memory and Instruction Buffer Performance" by 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" by T. Kilburn, D.B.G. Edwards, M.J. Lanigan, 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" by 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" by 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.*

[SS10] "FlashVM: Virtual Memory Management on Flash" by Mohit Saxena, Michael M. Swift. USENIX ATC '10, June, 2010, Boston, MA. *An early, excellent paper by our colleagues at U. Wisconsin about how to use Flash for paging. One interesting twist is how the system has to take* ***wearout****, an intrinsic property of Flash-based devices, into account. Read more about Flash-based SSDs later in this book if you are interested.*

[W+21] "The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus" by Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau. FAST '21, held virtually – thanks, COVID-19. *Our own work on a caching approach on modern devices; is it directly related to page replacement? Perhaps not. But it is a fun paper.*

Homework (Simulation)

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

Complete Virtual Memory Systems

Before we end our study of virtualizing memory, let us take a closer look at how entire virtual memory systems are put together. We've seen key elements of such systems, including numerous page-table designs, interactions with the TLB (sometimes, even handled by the OS itself), and strategies for deciding which pages to keep in memory and which to kick out. However, there are many other features that comprise a complete virtual memory system, including numerous features for performance, functionality, and security. And thus, our crux:

> THE CRUX: HOW TO BUILD A COMPLETE VM SYSTEM
>
> What features are needed to realize a complete virtual memory system? How do they improve performance, increase security, or otherwise improve the system?

We'll do this by covering two systems. The first is one of the earliest examples of a "modern" virtual memory manager, that found in the **VAX/VMS** operating system [LL82], as developed in the 1970's and early 1980's; a surprising number of techniques and approaches from this system survive to this day, and thus it is well worth studying. Some ideas, even those that are 50 years old, are still worth knowing, a thought that is well known to those in most other fields (e.g., Physics), but has to be stated in technology-driven disciplines (e.g., Computer Science).

The second is that of **Linux**, for reasons that should be obvious. Linux is a widely used system, and runs effectively on systems as small and underpowered as phones to the most scalable multicore systems found in modern datacenters. Thus, its VM system must be flexible enough to run successfully in all of those scenarios. We will discuss each system to illustrate how concepts brought forth in earlier chapters come together in a complete memory manager.

23.1 VAX/VMS Virtual Memory

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.

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.

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 ensure 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;

ASIDE: 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. It is no less real today, where Linux is expected to run well on your phone, a TV set-top box, a laptop computer, desktop computer, and a high-end server running thousands of processes in a cloud-based datacenter.

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 process); 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.

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.

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 con-

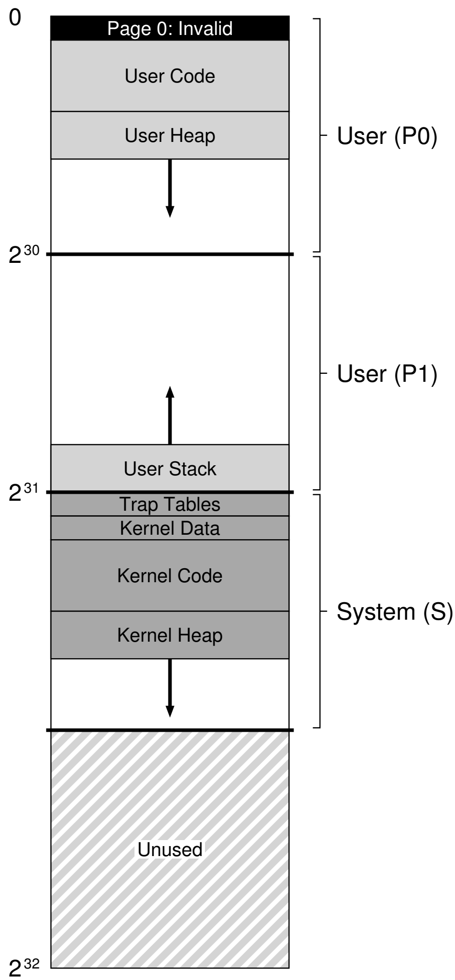


Figure 23.1: **The VAX/VMS Address Space**

text 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

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 10 to virtual addr 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).

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.

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.

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.

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].

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.

Other Neat 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 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 such work is avoided, and thus the virtue of demand zeroing.

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

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!

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.

23.2 The Linux Virtual Memory System

We'll now discuss some of the more interesting aspects of the Linux VM system. Linux development has been driven forward by real engineers solving real problems encountered in production, and thus a large number of features have slowly been incorporated into what is now a fully functional, feature-filled virtual memory system.

While we won't be able to discuss *every* aspect of Linux VM, we'll touch on the most important ones, especially where it has gone beyond what is found in classic VM systems such as VAX/VMS. We'll also try to highlight commonalities between Linux and older systems.

For this discussion, we'll focus on Linux for Intel x86. While Linux can and does run on many different processor architectures, Linux on x86 is its most dominant and important deployment, and thus the focus of our attention.

The Linux Address Space

Much like other modern operating systems, and also like VAX/VMS, a Linux virtual address space[1] consists of a user portion (where user

[1]Until recent changes, due to security threats, that is. Read the subsections below about Linux security for details on this modification.

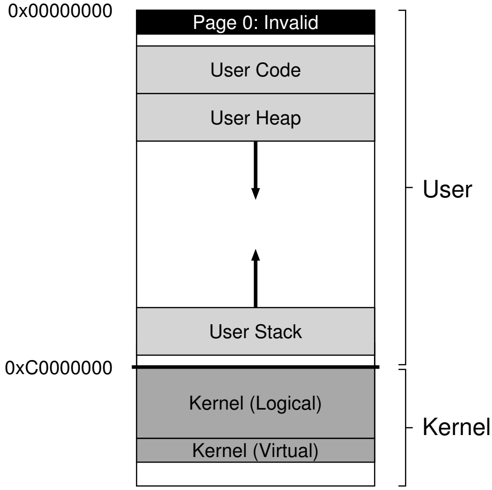


Figure 23.2: **The Linux Address Space**

program code, stack, heap, and other parts reside) and a kernel portion (where kernel code, stacks, heap, and other parts reside). Like those other systems, upon a context switch, the user portion of the currently-running address space changes; the kernel portion is the same across processes. Like those other systems, a program running in user mode cannot access kernel virtual pages; only by trapping into the kernel and transitioning to privileged mode can such memory be accessed.

In classic 32-bit Linux (i.e., Linux with a 32-bit virtual address space), the split between user and kernel portions of the address space takes place at address `0xC0000000`, or three-quarters of the way through the address space. Thus, virtual addresses `0` through `0xBFFFFFFF` are user virtual addresses; the remaining virtual addresses (`0xC0000000` through `0xFFFFFFFF`) are in the kernel's virtual address space. 64-bit Linux has a similar split but at slightly different points. Figure 23.2 shows a depiction of a typical (simplified) address space.

One slightly interesting aspect of Linux is that it contains two types of kernel virtual addresses. The first are known as **kernel logical addresses** [O16]. This is what you would consider the normal virtual address space of the kernel; to get more memory of this type, kernel code merely needs to call **`kmalloc`**. Most kernel data structures live here, such as page tables, per-process kernel stacks, and so forth. Unlike most other memory in the system, kernel logical memory *cannot* be swapped to disk.

The most interesting aspect of kernel logical addresses is their connection to physical memory. Specifically, there is a direct mapping between kernel logical addresses and the first portion of physical memory. Thus, kernel logical address `0xC0000000` translates to physical address

`0x00000000`, `0xC0000FFF` to `0x00000FFF`, and so forth. This direct mapping has two implications. The first is that it is simple to translate back and forth between kernel logical addresses and physical addresses; as a result, these addresses are often treated as if they are indeed physical. The second is that if a chunk of memory is contiguous in kernel logical address space, it is also contiguous in physical memory. This makes memory allocated in this part of the kernel's address space suitable for operations which need contiguous physical memory to work correctly, such as I/O transfers to and from devices via **directory memory access** (**DMA**) (something we'll learn about in the third part of this book).

The other type of kernel address is a **kernel virtual address**. To get memory of this type, kernel code calls a different allocator, **`vmalloc`**, which returns a pointer to a virtually contiguous region of the desired size. Unlike kernel logical memory, kernel virtual memory is usually not contiguous; each kernel virtual page may map to non-contiguous physical pages (and is thus not suitable for DMA). However, such memory is easier to allocate as a result, and thus used for large buffers where finding a contiguous large chunk of physical memory would be challenging.

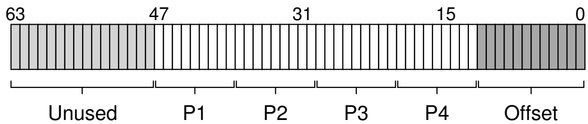
In 32-bit Linux, one other reason for the existence of kernel virtual addresses is that they enable the kernel to address more than (roughly) 1 GB of memory. Years ago, machines had much less memory than this, and enabling access to more than 1 GB was not an issue. However, technology progressed, and soon there was a need to enable the kernel to use larger amounts of memory. Kernel virtual addresses, and their disconnection from a strict one-to-one mapping to physical memory, make this possible. However, with the move to 64-bit Linux, the need is less urgent, because the kernel is not confined to only the last 1 GB of the virtual address space.

Page Table Structure

Because we are focused on Linux for x86, our discussion will center on the type of page-table structure provided by x86, as it determines what Linux can and cannot do. As mentioned before, x86 provides a hardware-managed, multi-level page table structure, with one page table per process; the OS simply sets up mappings in its memory, points a privileged register at the start of the page directory, and the hardware handles the rest. The OS gets involved, as expected, at process creation, deletion, and upon context switches, making sure in each case that the correct page table is being used by the hardware MMU to perform translations.

Probably the biggest change in recent years is the move from 32-bit x86 to 64-bit x86, as briefly mentioned above. As seen in the VAX/VMS system, 32-bit address spaces have been around for a long time, and as technology changed, they were finally starting to become a real limit for programs. Virtual memory makes it easy to program systems, but with modern systems containing many GB of memory, 32 bits were no longer enough to refer to each of them. Thus, the next leap became necessary.

Moving to a 64-bit address affects page table structure in x86 in the expected manner. Because x86 uses a multi-level page table, current 64-bit systems use a four-level table. The full 64-bit nature of the virtual address space is not yet in use, however, rather only the bottom 48 bits. Thus, a virtual address can be viewed as follows:



As you can see in the picture, the top 16 bits of a virtual address are unused (and thus play no role in translation), the bottom 12 bits (due to the 4-KB page size) are used as the offset (and hence just used directly, and not translated), leaving the middle 36 bits of virtual address to take part in the translation. The P1 portion of the address is used to index into the topmost page directory, and the translation proceeds from there, one level at a time, until the actual page of the page table is indexed by P4, yielding the desired page table entry.

As system memories grow even larger, more parts of this voluminous address space will become enabled, leading to five-level and eventually six-level page-table tree structures. Imagine that: a simple page table lookup requiring six levels of translation, just to figure out where in memory a certain piece of data resides.

Large Page Support

Intel x86 allows for the use of multiple page sizes, not just the standard 4-KB page. Specifically, recent designs support 2-MB and even 1-GB pages in hardware. Thus, over time, Linux has evolved to allow applications to utilize these **huge pages** (as they are called in the world of Linux).

Using huge pages, as hinted at earlier, leads to numerous benefits. As seen in VAX/VMS, doing so reduces the number of mappings that are needed in the page table; the larger the pages, the fewer the mappings. However, fewer page-table entries is not the driving force behind huge pages; rather, it's better TLB behavior and related performance gains.

When a process actively uses a large amount of memory, it quickly fills up the TLB with translations. If those translations are for 4-KB pages, only a small amount of total memory can be accessed without inducing TLB misses. The result, for modern "big memory" workloads running on machines with many GBs of memory, is a noticeable performance cost; recent research shows that some applications spend 10% of their cycles servicing TLB misses [B+13].

Huge pages allow a process to access a large tract of memory without TLB misses, by using fewer slots in the TLB, and thus is the main advantage. However, there are other benefits to huge pages: there is a shorter TLB-miss path, meaning that when a TLB miss does occur, it is

TIP: CONSIDER INCREMENTALISM

Many times in life, you are encouraged to be a revolutionary. "Think big!", they say. "Change the world!", they scream. And you can see why it is appealing; in some cases, big changes are needed, and thus pushing hard for them makes a lot of sense. And, if you try it this way, at least they might stop yelling at you.

However, in many cases, a slower, more incremental approach might be the right thing to do. The Linux huge page example in this chapter is an example of engineering incrementalism; instead of taking the stance of a fundamentalist and insisting large pages were the way of the future, developers took the measured approach of first introducing specialized support for it, learning more about its upsides and downsides, and, only when there was real reason for it, adding more generic support for all applications.

Incrementalism, while sometimes scorned, often leads to slow, thoughtful, and sensible progress. When building systems, such an approach might just be the thing you need. Indeed, this may be true in life as well.

serviced more quickly. In addition, allocation can be quite fast (in certain scenarios), a small but sometimes important benefit.

One interesting aspect of Linux support for huge pages is how it was done incrementally. At first, Linux developers knew such support was only important for a few applications, such as large databases with stringent performance demands. Thus, the decision was made to allow applications to explicitly request memory allocations with large pages (either through the `mmap()` or `shmget()` calls). In this way, most applications would be unaffected (and continue to use only 4-KB pages; a few demanding applications would have to be changed to use these interfaces, but for them it would be worth the pain.

More recently, as the need for better TLB behavior is more common among many applications, Linux developers have added **transparent** huge page support. When this feature is enabled, the operating system automatically looks for opportunities to allocate huge pages (usually 2 MB, but on some systems, 1 GB) without requiring application modification.

Huge pages are not without their costs. The biggest potential cost is **internal fragmentation**, i.e., a page that is large but sparsely used. This form of waste can fill memory with large but little used pages. Swapping, if enabled, also does not work well with huge pages, sometimes greatly amplifying the amount of I/O a system does. Overhead of allocation can also be bad (in some other cases). Overall, one thing is clear: the 4-KB page size which served systems so well for so many years is not the universal solution it once was; growing memory sizes demand that we consider large pages and other solutions as part of a necessary evolution of VM systems. Linux's slow adoption of this hardware-based technology is evidence of the coming change.

The Page Cache

To reduce costs of accessing persistent storage (the focus of the third part of this book), most systems use aggressive **caching** subsystems to keep popular data items in memory. Linux, in this regard, is no different than traditional operating systems.

The Linux **page cache** is unified, keeping pages in memory from three primary sources: **memory-mapped files**, file data and metadata from devices (usually accessed by directing `read()` and `write()` calls to the file system), and heap and stack pages that comprise each process (sometimes called **anonymous memory**, because there is no named file underneath of it, but rather swap space). These entities are kept in a **page cache hash table**, allowing for quick lookup when said data is needed.

The page cache tracks if entries are **clean** (read but not updated) or **dirty** (a.k.a., **modified**). Dirty data is periodically written to the backing store (i.e., to a specific file for file data, or to swap space for anonymous regions) by background threads (called `pdflush`), thus ensuring that modified data eventually is written back to persistent storage. This background activity either takes place after a certain time period or if too many pages are considered dirty (both configurable parameters).

In some cases, a system runs low on memory, and Linux has to decide which pages to kick out of memory to free up space. To do so, Linux uses a modified form of **2Q** replacement [JS94], which we describe here.

The basic idea is simple: standard LRU replacement is effective, but can be subverted by certain common access patterns. For example, if a process repeatedly accesses a large file (especially one that is nearly the size of memory, or larger), LRU will kick every other file out of memory. Even worse: retaining portions of this file in memory isn't useful, as they are never re-referenced before getting kicked out of memory.

The Linux version of the 2Q replacement algorithm solves this problem by keeping two lists, and dividing memory between them. When accessed for the first time, a page is placed on one queue (called `A1` in the original paper, but the **inactive list** in Linux); when it is re-referenced, the page is promoted to the other queue (called `Aq` in the original, but the **active list** in Linux). When replacement needs to take place, the candidate for replacement is taken from the inactive list. Linux also periodically moves pages from the bottom of the active list to the inactive list, keeping the active list to about two-thirds of the total page cache size [G04].

Linux would ideally manage these lists in perfect LRU order, but, as discussed in earlier chapters, doing so is costly. Thus, as with many OSes, an approximation of LRU (similar to **clock** replacement) is used.

This 2Q approach generally behaves quite a bit like LRU, but notably handles the case where a cyclic large-file access occurs by confining the pages of that cyclic access to the inactive list. Because said pages are never re-referenced before getting kicked out of memory, they do not flush out other useful pages found in the active list.

ASIDE: THE UBIQUITY OF MEMORY-MAPPING

Memory mapping predates Linux by some years, and is used in many places within Linux and other modern systems. The idea is simple: by calling `mmap()` on an already opened file descriptor, a process is returned a pointer to the beginning of a region of virtual memory where the contents of the file seem to be located. By then using that pointer, a process can access any part of the file with a simple pointer dereference.

Accesses to parts of a memory-mapped file that have not yet been brought into memory trigger **page faults**, at which point the OS will page in the relevant data and make it accessible by updating the page table of the process accordingly (i.e., **demand paging**).

Every regular Linux process uses memory-mapped files, even the code in `main()` does not call `mmap()` directly, because of how Linux loads code from the executable and shared library code into memory. Below is the (highly abbreviated) output of the `pmap` command line tool, which shows what different mapping comprise the virtual address space of a running program (the shell, in this example, `tcsh`). The output shows four columns: the virtual address of the mapping, its size, the protection bits of the region, and the source of the mapping:

```
0000000000400000    372K r-x-- tcsh
00000000019d5000   1780K rw---   [anon ]
00007f4e7cf06000   1792K r-x-- libc-2.23.so
00007f4e7d2d0000     36K r-x-- libcrypt-2.23.so
00007f4e7d508000    148K r-x-- libtinfo.so.5.9
00007f4e7d731000    152K r-x-- ld-2.23.so
00007f4e7d932000     16K rw---   [stack ]
```

As you can see from this output, the code from the `tcsh` binary, as well as code from `libc`, `libcrypt`, `libtinfo`, and code from the dynamic linker itself (`ld.so`) are all mapped into the address space. Also present are two anonymous regions, the heap (the second entry, labeled `anon`) and the stack (labeled `stack`). Memory-mapped files provide a straightforward and efficient way for the OS to construct a modern address space.

Security And Buffer Overflows

Probably the biggest difference between modern VM systems (Linux, Solaris, or one of the BSD variants) and ancient ones (VAX/VMS) is the emphasis on security in the modern era. Protection has always been a serious concern for operating systems, but with machines more interconnected than ever, it is no surprise that developers have implemented a variety of defensive countermeasures to halt those wily hackers from gaining control of systems.

One major threat is found in **buffer overflow** attacks[2], which can be used against normal user programs and even the kernel itself. The idea of these attacks is to find a bug in the target system which lets the attacker inject arbitrary data into the target's address space. Such vulnerabilities sometime arise because the developer assumes (erroneously) that an input will not be overly long, and thus (trustingly) copies the input into a buffer; because the input is in fact too long, it overflows the buffer, thus overwriting memory of the target. Code as innocent as the below can be the source of the problem:

```
int some_function(char *input) {
  char dest_buffer[100];
  strcpy(dest_buffer, input); // oops, unbounded copy!
}
```

In many cases, such an overflow is not catastrophic, e.g., bad input innocently given to a user program or even the OS will probably cause it to crash, but no worse. However, malicious programmers can carefully craft the input that overflows the buffer so as to inject their own code into the targeted system, essentially allowing them to take it over and do their own bidding. If successful upon a network-connected user program, attackers can run arbitrary computations or even rent out cycles on the compromised system; if successful upon the operating system itself, the attack can access even more resources, and is a form of what is called **privilege escalation** (i.e., user code gaining kernel access rights). If you can't guess, these are all Bad Things.

The first and most simple defense against buffer overflow is to prevent execution of any code found within certain regions of an address space (e.g., within the stack). The **NX bit** (for No-eXecute), introduced by AMD into their version of x86 (a similar XD bit is now available on Intel's), is one such defense; it just prevents execution from any page which has this bit set in its corresponding page table entry. The approach prevents code, injected by an attacker into the target's stack, from being executed, and thus mitigates the problem.

However, clever attackers are ... clever, and even when injected code cannot be added explicitly by the attacker, arbitrary code sequences can be executed by malicious code. The idea is known, in its most general form, as a **return-oriented programming** (**ROP**) [S07], and really it is quite brilliant. The observation behind ROP is that there are lots of bits of code (**gadgets**, in ROP terminology) within any program's address space, especially C programs that link with the voluminous C library. Thus, an attacker can overwrite the stack such that the return address in the currently executing function points to a desired malicious instruction (or

[2]See https://en.wikipedia.org/wiki/Buffer_overflow for some details and links about this topic, including a reference to the famous article by the security hacker Elias Levy, also known as "Aleph One".

series of instructions), followed by a return instruction. By stringing together a large number of gadgets (i.e., ensuring each return jumps to the next gadget), the attacker can execute arbitrary code. Amazing!

To defend against ROP (including its earlier form, the **return-to-libc attack** [S+04]), Linux (and other systems) add another defense, known as **address space layout randomization** (**ASLR**). Instead of placing code, stack, and the heap at fixed locations within the virtual address space, the OS randomizes their placement, thus making it quite challenging to craft the intricate code sequence required to implement this class of attacks. Most attacks on vulnerable user programs will thus cause crashes, but not be able to gain control of the running program.

Interestingly, you can observe this randomness in practice rather easily. Here's a piece of code that demonstrates it on a modern Linux system:

```
int main(int argc, char *argv[]) {
    int stack = 0;
    printf("%p\n", &stack);
    return 0;
}
```

This code just prints out the (virtual) address of a variable on the stack. In older non-ASLR systems, this value would be the same each time. But, as you can see below, the value changes with each run:

```
prompt> ./random
0x7ffd3e55d2b4
prompt> ./random
0x7ffe1033b8f4
prompt> ./random
0x7ffe45522e94
```

ASLR is such a useful defense for user-level programs that it has also been incorporated into the kernel, in a feature unimaginatively called **kernel address space layout randomization** (**KASLR**). However, it turns out the kernel may have even bigger problems to handle, as we discuss next.

Other Security Problems: Meltdown And Spectre

As we write these words (August, 2018), the world of systems security has been turned upside down by two new and related attacks. The first is called **Meltdown**, and the second **Spectre**. They were discovered at about the same time by four different groups of researchers/engineers, and have led to deep questioning of the fundamental protections offered by computer hardware and the OS above. See `meltdownattack.com` and `spectreattack.com` for papers describing each attack in detail. Spectre is considered the more problematic of the two.

The general weakness exploited in each of these attacks is that the CPUs found in modern systems perform all sorts of crazy behind-the-scenes tricks to improve performance. One class of technique that lies at the core of the problem is called **speculative execution**, in which the CPU guesses which instructions will soon be executed in the future, and starts executing them ahead of time. If the guesses are correct, the program runs faster; if not, the CPU undoes their effects on architectural state (e.g., registers) tries again, this time going down the right path.

The problem with speculation is that it tends to leave traces of its execution in various parts of the system, such as processor caches, branch predictors, etc. And thus the problem: as the authors of the attacks show, such state can make vulnerable the contents of memory, even memory that we thought was protected by the MMU.

One avenue to increasing kernel protection was thus to remove as much of the kernel address space from each user process and instead have a separate kernel page table for most kernel data (called **kernel page-table isolation**, or **KPTI**) [G+17]. Thus, instead of mapping the kernel's code and data structures into each process, only the barest minimum is kept therein; when switching into the kernel, then, a switch to the kernel page table is now needed. Doing so improves security and avoids some attack vectors, but at a cost: performance. Switching page tables is costly. Ah, the costs of security: convenience *and* performance.

Unfortunately, KPTI doesn't solve all of the security problems laid out above, just some of them. And simple solutions, such as turning off speculation, would make little sense, because systems would run thousands of times slower. Thus, it is an interesting time to be alive, if systems security is your thing.

To truly understand these attacks, you'll (likely) have to learn a lot more first. Begin by understanding modern computer architecture, as found in advanced books on the topic, focusing on speculation and all the mechanisms needed to implement it. Definitely read about the Meltdown and Spectre attacks, at the websites mentioned above; they actually also include a useful primer on speculation, so perhaps are not a bad place to start. And study the operating system for further vulnerabilities. Who knows what problems remain?

23.3 Summary

You have now seen a top-to-bottom review of two virtual memory systems. Hopefully, most of the details were easy to follow, as you should have already had a good understanding of the basic mechanisms and policies. More detail on VAX/VMS is available in the excellent (and short) paper by Levy and Lipman [LL82]. We encourage you to read it, as it is a great way to see what the source material behind these chapters is like.

You have also learned a bit about Linux. While a large and complex system, it inherits many good ideas from the past, many of which we have not had room to discuss in detail. For example, Linux performs lazy copy-on-write copying of pages upon `fork()`, thus lowering overheads by avoiding unnecessary copying. Linux also demand zeroes pages (using memory-mapping of the `/dev/zero` device), and has a background swap daemon (**`swapd`**) that swaps pages to disk to reduce memory pressure. Indeed, the VM is filled with good ideas taken from the past, and also includes many of its own innovations.

To learn more, check out these reasonable (but, alas, outdated) books [BC05,G04]. We encourage you to read them on your own, as we can only provide the merest drop from what is an ocean of complexity. But, you've got to start somewhere. What is any ocean, but a multitude of drops? [M04]

References

[B+13] "Efficient Virtual Memory for Big Memory Servers" by A. Basu, J. Gandhi, J. Chang, M. D. Hill, M. M. Swift. ISCA '13, June 2013, Tel-Aviv, Israel. *A recent work showing that TLBs matter, consuming 10% of cycles for large-memory workloads. The solution: one massive segment to hold large data sets. We go backward, so that we can go forward!*

[BB+72] "TENEX, A Paged Time Sharing System for the PDP-10" by D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, R. S. Tomlinson. CACM, 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; also an inspiration for other aspects of modern systems, including process management, virtual memory, and file systems.*

[BJ81] "Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Reference Bits" by O. Babaoglu, W. N. Joy. SOSP '81, Pacific Grove, California, December 1981. *How to exploit existing protection machinery to emulate reference bits, from a group at Berkeley working on their own version of* UNIX*: the* ***Berkeley Systems Distribution (BSD)****. The group was influential in the development of virtual memory, file systems, and networking.*

[BC05] "Understanding the Linux Kernel" by D. P. Bovet, M. Cesati. O'Reilly Media, November 2005. *One of the many books you can find on Linux, which are out of date, but still worthwhile.*

[C03] "The Innovator's Dilemma" by 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" by H. Custer, D. 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.*

[G04] "Understanding the Linux Virtual Memory Manager" by M. Gorman. Prentice Hall, 2004. *An in-depth look at Linux VM, but alas a little out of date.*

[G+17] "KASLR is Dead: Long Live KASLR" by D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, S. Mangard. Engineering Secure Software and Systems, 2017. Available: `https://gruss.cc/files/kaiser.pdf` *Excellent info on KASLR, KPTI, and beyond.*

[JS94] "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm" by T. Johnson, D. Shasha. VLDB '94, Santiago, Chile. *A simple but effective approach to building page replacement.*

[LL82] "Virtual Memory Management in the VAX/VMS Operating System" by H. Levy, P. Lipman. IEEE Computer, Volume 15:3, March 1982. *Read the original source of most of this material. 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.*

[M04] "Cloud Atlas" by D. Mitchell. Random House, 2004. *It's hard to pick a favorite book. There are too many! Each is great in its own unique way. But it'd be hard for these authors not to pick "Cloud Atlas", a fantastic, sprawling epic about the human condition, from where the the last quote of this chapter is lifted. If you are smart – and we think you are – you should stop reading obscure commentary in the references and instead read "Cloud Atlas"; you'll thank us later.*

[O16] "Virtual Memory and Linux" by A. Ott. Embedded Linux Conference, April 2016. https://events.static.linuxfound.org/sites/events/files/slides/elc_2016_mem.pdf . *A useful set of slides which gives an overview of the Linux VM.*

[RL81] "Segmented FIFO Page Replacement" by R. Turner, H. Levy. SIGMETRICS '81, Las Vegas, Nevada, September 1981. *A short paper that shows for some workloads, segmented FIFO can approach the performance of LRU.*

[S07] "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)" by H. Shacham. CCS '07, October 2007. *A generalization of return-to-libc. Dr. Beth Garner said in Basic Instinct, "She's crazy! She's brilliant!" We might say the same about ROP.*

[S+04] "On the Effectiveness of Address-space Randomization" by H. Shacham, M. Page, B. Pfaff, E. J. Goh, N. Modadugu, D. Boneh. CCS '04, October 2004. *A description of the return-to-libc attack and its limits. Start reading, but be wary: the rabbit hole of systems security is deep...*

24

Summary Dialogue on Memory Virtualization

Student: *(Gulps) Wow, that was a lot of material.*

Professor: *Yes, and?*

Student: *Well, how am I supposed to remember it all? You know, for the exam?*

Professor: *Goodness, I hope that's not why you are trying to remember it.*

Student: *Why should I then?*

Professor: *Come on, I thought you knew better. You're trying to learn something here, so that when you go off into the world, you'll understand how systems actually work.*

Student: *Hmm... can you give an example?*

Professor: *Sure! One time back in graduate school, my friends and I were measuring how long memory accesses took, and once in a while the numbers were way higher than we expected; we thought all the data was fitting nicely into the second-level hardware cache, you see, and thus should have been really fast to access.*

Student: *(nods)*

Professor: *We couldn't figure out what was going on. So what do you do in such a case? Easy, ask a professor! So we went and asked one of our professors, who looked at the graph we had produced, and simply said "TLB". Aha! Of course, TLB misses! Why didn't we think of that? Having a good model of how virtual memory works helps diagnose all sorts of interesting performance problems.*

Student: *I think I see. I'm trying to build these mental models of how things work, so that when I'm out there working on my own, I won't be surprised when a system doesn't quite behave as expected. I should even be able to anticipate how the system will work just by thinking about it.*

Professor: *Exactly. So what have you learned? What's in your mental model of how virtual memory works?*

Student: *Well, I think I now have a pretty good idea of what happens when memory is referenced by a process, which, as you've said many times, happens on each instruction fetch as well as explicit loads and stores.*

Professor: *Sounds good — tell me more.*

Student: *Well, one thing I'll always remember is that the addresses we see in a user program, written in C for example...*

Professor: *What other language is there?*

Student: *(continuing) ... Yes, I know you like C. So do I! Anyhow, as I was saying, I now really know that all addresses that we can observe within a program are virtual addresses; that I, as a programmer, am just given this illusion of where data and code are in memory. I used to think it was cool that I could print the address of a pointer, but now I find it frustrating — it's just a virtual address! I can't see the real physical address where the data lives.*

Professor: *Nope, the OS definitely hides that from you. What else?*

Student: *Well, I think the TLB is a really key piece, providing the system with a small hardware cache of address translations. Page tables are usually quite large and hence live in big and slow memories. Without that TLB, programs would certainly run a great deal more slowly. Seems like the TLB truly makes virtualizing memory possible. I couldn't imagine building a system without one! And I shudder at the thought of a program with a working set that exceeds the coverage of the TLB: with all those TLB misses, it would be hard to watch.*

Professor: *Yes, cover the eyes of the children! Beyond the TLB, what did you learn?*

Student: *I also now understand that the page table is one of those data structures you need to know about; it's just a data structure, though, and that means almost any structure could be used. We started with simple structures, like arrays (a.k.a. linear page tables), and advanced all the way up to multi-level tables (which look like trees), and even crazier things like pageable page tables in kernel virtual memory. All to save a little space in memory!*

Professor: *Indeed.*

Student: *And here's one more important thing: I learned that the address translation structures need to be flexible enough to support what programmers want to do with their address spaces. Structures like the multi-level table are perfect in this sense; they only create table space when the user needs a portion of the address space, and thus there is little waste. Earlier attempts, like the simple base and bounds register, just weren't flexible enough; the structures need to match what users expect and want out of their virtual memory system.*

Professor: *That's a nice perspective. What about all of the stuff we learned about swapping to disk?*

Student: *Well, it's certainly fun to study, and good to know how page replacement works. Some of the basic policies are kind of obvious (like LRU, for example), but building a real virtual memory system seems more interesting, like* *we saw in the VMS case study. But somehow, I found the mechanisms more interesting, and the policies less so.*

Professor: *Oh, why is that?*

Student: *Well, as you said, in the end the best solution to policy problems is simple: buy more memory. But the mechanisms you need to understand to know how stuff really works. Speaking of which...*

Professor: *Yes?*

Student: *Well, my machine is running a little slowly these days... and memory certainly doesn't cost that much...*

Professor: *Oh fine, fine! Here's a few bucks. Go and get yourself some DRAM, cheapskate.*

Student: *Thanks professor! I'll never swap to disk again — or, if I do, at least I'll know what's actually going on!*

52

A Dialogue on Security

Chapter by **Peter Reiher (UCLA)**

Professor: *Hello again, student!*

Student: *I thought we were done with all this. We've already had three pillars, and I even stuck around for a few appendices. Will I never be done with this class?*

Professor: *That depends on who I am. Some professors want to talk about security and some don't. Unfortunately for you, given that you're here, I'm one of those who want to.*

Student: *OK, I suppose we'd better just get on with it.*

Professor: *That's the spirit! Soonest begun, soonest done. So, let's say you have a peach...*

Student: *You told me we were at least done with peaches!*

Professor: *When one is discussing security, lies will always be a part of the discussion. Anyway, you've got a peach. You certainly wouldn't want to turn around and find someone had stolen your peach, would you?*

Student: *Well, if it isn't as rotten as the one you ended up with, I suppose not.*

Professor: *And you probably wouldn't be any happier if you turned around and discovered someone had swapped out your peach for a turnip, either, would you?*

Student: *I guess not, though I do know a couple of good recipes for turnips.*

Professor: *And you also wouldn't want somebody slapping your hand away every time you reached for your peach, right?*

Student: *No, that would be pretty rude.*

Professor: *You wouldn't want that happening to any of the resources your computer controls, either. You might be even unhappier, if they're really important resources. You wouldn't want the love letter you're in the middle of composing to leak out, you wouldn't want someone to reset the saved state in your favorite game to take you back to the very beginning, and you would be mighty upset if, at midnight the evening before your project was due, you weren't allowed to log into your computer.*

Student: *True, those would all pretty much suck.*

Professor: *Let's try to keep a professional tone here. After all, this is a classroom. Kind of. That's what operating system security is all about, and that's what I'm here to tell you about. How can you ensure that secrets remain confidential? How can you guarantee the integrity of your important data? How can you ensure that you can use your computer resources when you want to? And these questions apply to all of the resources in your computer, all the time, forever.*

Student: *All this sounds a little like reliability stuff we talked about before...*

Professor: *Yes and no. Bad things can happen more or less by accident or through poor planning, and reliability is about those sorts of things. But we're going a step further. SOMEBODY WANTS YOUR PEACH!!!!*

Student: *Stop shouting! You were the one asking for a professional tone.*

Professor: *My apologies, I get excited about this stuff sometimes. The point I was trying to make is that when we talk about security, we're talking about genuine adversaries, human adversaries who are trying to make things go wrong for you. That has some big implications. They're likely to be clever, malevolent, persistent, flexible, and sneaky. You may already feel like the universe has it in for you (most students feel that way, at any rate), but these folks really, truly are out to get you. You're going to have to protect your assets despite anything they try.*

Student: *This sounds challenging.*

Professor: *You have no idea... But you will! YOU WILL!! (maniacal laughter)*

53

Introduction to Operating System Security

Chapter by **Peter Reiher (UCLA)**

53.1 Introduction

Security of computing systems is a vital topic whose importance only keeps increasing. Much money has been lost and many people's lives have been harmed when computer security has failed. Attacks on computer systems are so common as to be inevitable in almost any scenario where you perform computing. Generally, all elements of a computer system can be subject to attack, and flaws in any of them can give an attacker an opportunity to do something you want to prevent. But operating systems are particularly important from a security perspective. Why?

To begin with, pretty much everything runs on top of an operating system. As a rule, if the software you are running on top of, whether it be an operating system, a piece of middleware, or something else, is insecure, what's above it is going to also be insecure. It's like building a house on sand. You may build a nice solid structure, but a flood can still wash away the base underneath your home, totally destroying it despite the care you took in its construction. Similarly, your application might perhaps have no security flaws of its own, but if the attacker can misuse the software underneath you to steal your information, crash your program, or otherwise cause you harm, your own efforts to secure your code might be for naught.

This point is especially important for operating systems. You might not care about the security of a particular web server or database system if you don't run that software, and you might not care about the security of some middleware platform that you don't use, but everyone runs an operating system, and there are relatively few choices of which to run. Thus, security flaws in an operating system, especially a widely used one, have an immense impact on many users and many pieces of software.

Another reason that operating system security is so important is that ultimately all of our software relies on proper behavior of the underlying hardware: the processor, the memory, and the peripheral devices. What has ultimate control of those hardware resources? The operating system.

Thinking about what you have already studied concerning memory management, scheduling, file systems, synchronization, and so forth, what would happen with each of these components of your operating system if an adversary could force it to behave in some arbitrarily bad way? If you understand what you've learned so far, you should find this prospect deeply disturbing[1]. Our computing lives depend on our operating systems behaving as they have been defined to behave, and particularly on them not behaving in ways that benefit our adversaries, rather than us.

The task of securing an operating system is not an easy one, since modern operating systems are large and complex. Your experience in writing code should have already pointed out to you that the more code you've got, and the more complex the algorithms are, the more likely your code is to contain flaws. Failures in software security generally arise from these kinds of flaws. Large, complex programs are likely to be harder to secure than small, simple programs. Not many other programs are as large and complex as a modern operating system.

Another challenge in securing operating systems is that they are, for the most part, meant to support multiple processes simultaneously. As you've learned, there are many mechanisms in an operating system meant to segregate processes from each other, and to protect shared pieces of hardware from being used in ways that interfere with other processes. If every process could be trusted to do anything it wants with any hardware resource and any piece of data on the machine without harming any other process, securing the system would be a lot easier. However, we typically don't trust everything equally. When you download and run a script from a web site you haven't visited before, do you really want it to be able to wipe every file from your disk, kill all your other processes, and start using your network interface to send spam email to other machines? Probably not, but if you are the owner of your computer, you have the right to do all those things, if that's what you want to do. And unless the operating system is careful, any process it runs, including the one running that script you downloaded, can do anything you can do.

Consider the issue of operating system security from a different perspective. One role of an operating system is to provide useful abstractions for application programs to build on. These applications must rely on the OS implementations of the abstractions to work as they are defined. Often, one part of the definition of such abstractions is their security behavior. For example, we expect that the operating system's file system will enforce the access restrictions it is supposed to enforce. Applications can then build on this expectation to achieve the security goals they require, such as counting on the file system access guarantees to ensure that a file they have specified as unwriteable does not get altered. If the applications cannot rely on proper implementation of security guarantees for OS abstractions, then they cannot use these abstractions to achieve their own security goals. At the minimum, that implies a great deal more work on

[1]If you don't understand it, you have a lot of re-reading to do. A lot.

the part of the application developers, since they will need to take extra measures to achieve their desired security goals. Taking into account our earlier discussion, they will often be unable to achieve these goals if the abstractions they must rely on (such as virtual memory or a well-defined scheduling policy) cannot be trusted.

Obviously, operating system security is vital, yet hard to achieve. So what do we do to secure our operating system? Addressing that question has been a challenge for generations of computer scientists, and there is as yet no complete answer. But there are some important principles and tools we can use to secure operating systems. These are generally built into any general-purpose operating system you are likely to work with, and they alter what can be done with that system and how you go about doing it. So you might not think you're interested in security, but you need to understand what your OS does to secure itself to also understand how to get the system to do what you want.

CRUX: HOW TO SECURE OS RESOURCES

In the face of multiple possibly concurrent and interacting processes running on the same machine, how can we ensure that the resources each process is permitted to access are exactly those it should access, in exactly the ways we desire? 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 security?

53.2 What Are We Protecting?

We aren't likely to achieve good protection unless we have a fairly comprehensive view of what we're trying to protect when we say our operating system should be secure. Fortunately, that question is easy to answer for an operating system, at least at the high level: everything. That answer isn't very comforting, but it is best to have a realistic understanding of the broad implications of operating system security.

A typical commodity operating system has complete control of all (or almost all) hardware on the machine and is able to do literally anything the hardware permits. That means it can control the processor, read and write all registers, examine any main memory location, and perform any operation one of its peripherals supports. As a result, among the things the OS can do are:

- examine or alter any process's memory
- read, write, delete or corrupt any file on any writeable persistent storage medium, including hard disks and flash drives
- change the scheduling or even halt execution of any process
- send any message to anywhere, including altered versions of those a process wished to send
- enable or disable any peripheral device

ASIDE: SECURITY ENCLAVES

A little bit back, we said the operating system controls "almost all" the hardware on the machine. That kind of caveat should have gotten you asking, "well, what parts of the hardware doesn't it control?" Originally, it really was all the hardware. But starting in the 1990s, hardware developer began to see a need to keep some hardware isolated, to a degree, from the operating system. The first such hardware was primarily intended to protect the boot process of the operating system. **TPM**, or **Trusted Platform Module**, provided assurance that you were booting the version of the operating system you intended to, protecting you from attacks that tried to boot compromised versions of the system. More recently, more general hardware elements have tried to control what can be done on the machine, typically with some particularly important data, often data that is related to cryptography. Such hardware elements are called security enclaves, since they are meant to allow only safe use of this data, even by the most powerful, trusted code in the system the operating system itself. They are often used to support operations in a cloud computing environment, where multiple operating systems might be running under virtual machines sharing the same physical hardware.

This turns out to be a harder trick than anyone expected. Security tricks usually are. Security enclaves often prove not to provide quite as much isolation as their designers hoped. But the attacks on them tend to be sophisticated and difficult, and usually require the ability to run privileged code on the system already. So even if they don't achieve their full goals, they do put an extra protective barrier against compromised operating system code.

- give any process access to any other process's resources
- arbitrarily take away any resource a process controls
- respond to any system call with a maximally harmful lie

In essence, processes are at the mercy of the operating system. It is nearly impossible for a process to 'protect' any part of itself from a malicious operating system. We typically assume our operating system is not actually malicious[2], but a flaw that allows a malicious process to cause the operating system to misbehave is nearly as bad, since it could potentially allow that process to gain any of the powers of the operating system itself. This point should make you think very seriously about the importance of designing secure operating systems and, more commonly, applying security patches to any operating system you are running. Security flaws in your operating system can completely compromise everything about the machine the system runs on, so preventing them and patching any that are found is vitally important.

[2]If you suspect your operating system is malicious, it's time to get a new operating system.

53.3 Security Goals and Policies

What do we mean when we say we want an operating system, or any system, to be secure? That's a rather vague statement. What we really mean is that there are things we would like to happen in the system and things we don't want to happen, and we'd like a high degree of assurance that we get what we want. As in most other aspects of life, we usually end up paying for what we get, so it's worthwhile to think about exactly what security properties and effects we actually need and then pay only for those, not for other things we don't need. What this boils down to is that we want to specify the goals we have for the security-relevant behavior of our system and choose defense approaches likely to achieve those goals at a reasonable cost.

Researchers in security have thought about this issue in broad terms for a long time. At a high conceptual level, they have defined three big security-related goals that are common to many systems, including operating systems. They are:

- **Confidentiality** – If some piece of information is supposed to be hidden from others, don't allow them to find it out. For example, you don't want someone to learn what your credit card number is – you want that number kept confidential.

- **Integrity** – If some piece of information or component of a system is supposed to be in a particular state, don't allow an adversary to change it. For example, if you've placed an online order for delivery of one pepperoni pizza, you don't want a malicious prankster to change your order to 1000 anchovy pizzas. One important aspect of integrity is authenticity. It's often important to be sure not only that information has not changed, but that it was created by a particular party and not by an adversary.

- **Availability** – If some information or service is supposed to be available for your own or others' use, make sure an attacker cannot prevent its use. For example, if your business is having a big sale, you don't want your competitors to be able to block off the streets around your store, preventing your customers from reaching you.

An important extra dimension of all three of these goals is that we want controlled sharing in our systems. We share our secrets with some people and not with others. We allow some people to change our enterprise's databases, but not just anyone. Some systems need to be made available to a particular set of preferred users (such as those who have paid to play your on-line game) and not to others (who have not). Who's doing the asking matters a lot, in computers as in everyday life.

Another important aspect of security for computer systems is we often want to be sure that when someone told us something, they cannot later deny that they did so. This aspect is often called **non-repudiation**. The harder and more expensive it is for someone to repudiate their actions, the easier it is to hold them to account for those actions, and thus the less likely people are to perform malicious actions. After all, they might well get caught and will have trouble denying they did it.

These are big, general goals. For a real system, you need to drill down to more detailed, specific goals. In a typical operating system, for example, we might have a confidentiality goal stating that a process's memory space cannot be arbitrarily read by another process. We might have an integrity goal stating that if a user writes a record to a particular file, another user who should not be able to write that file can't change the record. We might have an availability goal stating that one process running on the system cannot hog the CPU and prevent other processes from getting their share of the CPU. If you think back on what you've learned about the process abstraction, memory management, scheduling, file systems, IPC, and other topics from this class, you should be able to think of some other obvious confidentiality, integrity, and availability goals we are likely to want in our operating systems.

For any particular system, even goals at this level are not sufficiently specific. The integrity goal alluded to above, where a user's file should not be overwritten by another user not permitted to do so, gives you a hint about the extra specificity we need in our security goals for a particular system. Maybe there is some user who should be able to overwrite the file, as might be the case when two people are collaborating on writing a report. But that doesn't mean an unrelated third user should be able to write that file, if he is not collaborating on the report stored there. We need to be able to specify such detail in our security goals. Operating systems are written to be used by many different people with many different needs, and operating system security should reflect that generality. What we want in security mechanisms for operating systems is flexibility in describing our detailed security goals.

Ultimately, of course, the operating system software must do its best to enforce those flexible security goals, which implies we'll need to encode those goals in forms that software can understand. We typically must convert our vague understandings of our security goals into highly specific **security policies**. For example, in the case of the file described above, we might want to specify a policy like 'users A and B may write to file X, but no other user can write it.' With that degree of specificity, backed by carefully designed and implemented mechanisms, we can hope to achieve our security goals.

Note an important implication for operating system security: in many cases, an operating system will have the mechanisms necessary to implement a desired security policy with a high degree of assurance in its proper application, but only if someone tells the operating system precisely what that policy is. With some important exceptions (like maintaining a process's address space private unless specifically directed otherwise), the operating system merely supplies general mechanisms that can implement many specific policies. Without intelligent design of poli-

ASIDE: SECURITY VS. FAULT TOLERANCE

When discussing the process abstraction, we talked about how virtualization protected a process from actions of other processes. For instance, we did not want our process's memory to be accidentally overwritten by another process, so our virtualization mechanisms had to prevent such behavior. Then we were talking primarily about flaws or mistakes in processes. Is this actually any different than worrying about malicious behavior, which is more commonly the context in which we discuss security? Have we already solved all our problems by virtualizing our resources?

Yes and no. (Isn't that a helpful phrase?) Yes, if we perfectly virtualized everything and allowed no interactions between anything, we very likely would have solved most problems of malice. However, most virtualization mechanisms are not totally bulletproof. They work well when no one tries to subvert them, but may not be perfect against all possible forms of misbehavior. Second, and perhaps more important, we don't really want to totally isolate processes from each other. Processes share some OS resources by default (such as file systems) and can optionally choose to share others. These intentional relaxations of virtualization are not problematic when used properly, but the possibilities of legitimate sharing they open are also potential channels for malicious attacks. Finally, the OS does not always have complete control of the hardware...

cies and careful application of the mechanisms, however, what the operating system *should* or *could* do may not be what your operating system *will* do.

53.4 Designing Secure Systems

Few of you will ever build your own operating system, nor even make serious changes to any existing operating system, but we expect many of you will build large software systems of some kind. Experience of many computer scientists with system design has shown that there are certain design principles that are helpful in building systems with security requirements. These principles were originally laid out by Jerome Saltzer and Michael Schroeder in an influential paper [SS75], though some of them come from earlier observations by others. While neither the original authors nor later commentators would claim that following them will guarantee that your system is secure, paying attention to them has proven to lead to more secure systems, while you ignore them at your own peril. We'll discuss them briefly here. If you are actually building a large software system, it would be worth your while to look up this paper (or more detailed commentaries on it) and study the concepts carefully.

1. **Economy of mechanism** – This basically means keep your system as small and simple as possible. Simple systems have fewer bugs and it's easier to understand their behavior. If you don't understand your system's behavior, you're not likely to know if it achieves its security goals.

2. **Fail-safe defaults** – Default to security, not insecurity. If policies can be set to determine the behavior of a system, have the default for those policies be more secure, not less.

3. **Complete mediation** – This is a security term meaning that you should check if an action to be performed meets security policies every single time the action is taken[3].

4. **Open design** – Assume your adversary knows every detail of your design. If the system can achieve its security goals anyway, you're in good shape. This principle does not necessarily mean that you actually tell everyone all the details, but base your security on the assumption that the attacker has learned everything. He often has, in practice.

5. **Separation of privilege** – Require separate parties or credentials to perform critical actions. For example, two-factor authentication, where you use both a password and possession of a piece of hardware to determine identity, is more secure than using either one of those methods alone.

6. **Least privilege** – Give a user or a process the minimum privileges required to perform the actions you wish to allow. The more privileges you give to a party, the greater the danger that they will abuse those privileges. Even if you are confident that the party is not malicious, if they make a mistake, an adversary can leverage their error to use their superfluous privileges in harmful ways.

7. **Least common mechanism** – For different users or processes, use separate data structures or mechanisms to handle them. For example, each process gets its own page table in a virtual memory system, ensuring that one process cannot access another's pages.

8. **Acceptability** – A critical property not dear to the hearts of many programmers. If your users won't use it, your system is worthless. Far too many promising secure systems have been abandoned because they asked too much of their users.

[3]This particular principle is often ignored in many systems, in favor of lower overhead or usability. An overriding characteristic of all engineering design is that you often must balance conflicting goals, as we saw earlier in the course, such as in the scheduling chapters. We'll say more about that in the context of security later.

These are not the only useful pieces of advice on designing secure systems out there. There is also lots of good material on taking the next step, converting a good design into code that achieves the security you intended, and other material on how to evaluate whether the system you have built does indeed meet those goals. These issues are beyond the scope of this course, but are extremely important when the time comes for you to build large, complex systems. For discussion of approaches to secure programming, you might start with Seacord [SE13], if you are working in C. If you are working in another language, you should seek out a similar text specific to that language, since many secure coding problem are related to details of the language. For a comprehensive treatment on how to evaluate if your system is secure, start with Dowd et al.'s work [D+07].

53.5 The Basics of OS Security

In a typical operating system, then, we have some set of security goals, centered around various aspects of confidentiality, integrity, and availability. Some of these goals tend to be built in to the operating system model, while others are controlled by the owners or users of the system. The built-in goals are those that are extremely common, or must be ensured to make the more specific goals achievable. Most of these built-in goals relate to controlling process access to pieces of the hardware. That's because the hardware is shared by all the processes on a system, and unless the sharing is carefully controlled, one process can interfere with the security goals of another process. Other built-in goals relate to services that the operating system offers, such as file systems, memory management, and interprocess communications. If these services are not carefully controlled, processes can subvert the system's security goals.

Clearly, a lot of system security is going to be related to process handling. If the operating system can maintain a clean separation of processes that can only be broken with the operating system's help, then neither shared hardware nor operating system services can be used to subvert our security goals. That requirement implies that the operating system needs to be careful about allowing use of hardware and of its services. In many cases, the operating system has good opportunities to apply such caution. For example, the operating system controls virtual memory, which in turn completely controls which physical memory addresses each process can access. Hardware support prevents a process from even naming a physical memory address that is not mapped into its virtual memory space. (The software folks among us should remember to regularly thank the hardware folks for all the great stuff they've given us to work with.)

System calls offer the operating system another opportunity to provide protection. In most operating systems, processes access system services by making an explicit system call, as was discussed in earlier chap-

TIP: BE CAREFUL OF THE WEAKEST LINK

It's worthwhile to remember that the people attacking your systems share many characteristics with you. In particular, they're probably pretty smart and they probably are kind of lazy, in the positive sense that they don't do work that they don't need to do. That implies that attackers tend to go for the easiest possible way to overcome your system's security. They're not going to search for a zero-day buffer overflow if you've chosen "password" as your password to access the system.

The practical implication for you is that you should spend most of the time you devote to securing your system to identifying and strengthening your weakest link. Your weakest link is the least protected part of your system, the one that's easiest to attack, the one you can't hide away or augment with some external security system. Often, a running system's weakest link is actually its human users, not its software. You will have a hard time changing the behavior of people, but you can design the software bearing in mind that attackers may try to fool the legitimate users into misusing it. Remember that principle of least privilege? If an attacker can fool a user who has complete privileges into misusing the system, it will be a lot worse than fooling a user who can only damage his own assets.

Generally, thinking about security is a bit different than thinking about many other system design issues. It's more adversarial. If you want to learn more about good ways to think about security of the systems you build, check out Schneier's book "Secrets and Lies" [SC00].

ters. As you have learned, system calls switch the execution mode from the processor's user mode to its supervisor mode, invoking an appropriate piece of operating system code as they do so. That code can determine which process made the system call and what service the process requested. Earlier, we only talked about how this could allow the operating system to call the proper piece of system code to perform the service, and to keep track of who to return control to when the service had been completed. But the same mechanism gives the operating system the opportunity to check if the requested service should be allowed under the system's security policy. Since access to peripheral devices is through device drivers, which are usually also accessed via system call, the same mechanism can ensure proper application of security policies for hardware access.

When a process performs a system call, then, the operating system will use the process identifier in the process control block or similar structure to determine the identity of the process. The OS can then use **access control mechanisms** to decide if the identified process is **authorized** to perform the requested action. If so, the OS either performs the action itself on behalf of the process or arranges for the process to perform it without further system intervention. If the process is not authorized, the OS can simply generate an error code for the system call and return control to the process, if the scheduling algorithm permits.

53.6 Summary

The security of the operating system is vital for both its own and its applications' sakes. Security failures in this software allow essentially limitless bad consequences. While achieving system security is challenging, there are known design principles that can help. These principles are useful not only in designing operating systems, but in designing any large software system.

Achieving security in operating systems depends on the security goals one has. These goals will typically include goals related to confidentiality, integrity, and availability. In any given system, the more detailed particulars of these security goals vary, which implies that different systems will have different security policies intended to help them meet their specific security goals. As in other areas of operating system design, we handle these varying needs by separating the specific policies used by any particular system from the general mechanisms used to implement the policies for all systems.

The next question to address is, what mechanisms should our operating system provide to help us support general security policies? The virtualization of processes and memory is one helpful mechanism, since it allows us to control the behavior of processes to a large extent. We will describe several other useful operating system security mechanisms in the upcoming chapters.

References

[D+07] "The Art of Software Security Assessment" by Mark Dowd, John McDonald, and Justin Schuh. Addison-Wesley, 2007. *A long, comprehensive treatment of how to determine if your software system meets its security goals. It also contains useful advice on avoiding security problems in coding.*

[SC00] "Secrets and Lies" by Bruce Schneier. Wiley Computer Publishing, 2000. *A good high-level perspective of the challenges of computer security, developed at book length. Intended for an audience of moderately technically sophisticated readers, and well regarded in the security community. A must-read if you intend to work in that field.*

[SE13] "Secure Coding in C and C++" by Robert Seacord. Addison-Wesley, 2013. *A well regarded book on how to avoid major security mistakes in coding in C.*

[SS75] "The Protection of Information in Computer Systems" by Jerome Saltzer and Michael Schroeder. Proceedings of the IEEE, Vol. 63, No. 9, September 1975. *A highly influential paper, particularly their codification of principles for secure system design.*

54

Authentication

Chapter by **Peter Reiher (UCLA)**

54.1 Introduction

Given that we need to deal with a wide range of security goals and security policies that are meant to achieve those goals, what do we need from our operating system? Operating systems provide services for processes, and some of those services have security implications. Clearly, the operating system needs to be careful in such cases to do the right thing, security-wise. But the reason operating system services are allowed at all is that sometimes they need to be done, so any service that the operating system might be able to perform probably should be performed – under the right circumstances.

Context will be everything in operating system decisions on whether to perform some service or to refuse to do so because it will compromise security goals. Perhaps the most important element of that context is who's doing the asking. In the real world, if your significant other asks you to pick up a gallon of milk at the store on the way home, you'll probably do so, while if a stranger on the street asks the same thing, you probably won't. In an operating system context, if the system administrator asks the operating system to install a new program, it probably should, while if a script downloaded from a random web page asks to install a new program, the operating system should take more care before performing the installation. In computer security discussions, we often refer to the party asking for something as the **principal**. Principals are security-meaningful entities that can request access to resources, such as human users, groups of users, or complex software systems.

So knowing who is requesting an operating system service is crucial in meeting your security goals. How does the operating system know that? Let's work a bit backwards here to figure it out.

Operating system services are most commonly requested by system calls made by particular processes, which trap from user code into the operating system. The operating system then takes control and performs some service in response to the system call. Associated with the calling process is the OS-controlled data structure that describes the process, so the operating system can check that data structure to determine the identity of the process. Based on that identity, the operating system now has the opportunity to make a policy-based decision on whether to perform the requested operation. In computer security discussions, the process or other active computing entity performing the request on behalf of a principal is often called its **agent**.

The request is for access to some particular resource, which we frequently refer to as the **object** of the access request[1]. Either the operating system has already determined this agent process can access the object or it hasn't. If it has determined that the process is permitted access, the OS can remember that decision and it's merely a matter of keeping track, presumably in some per-process data structure like the PCB, of that fact. For example, as we discovered when investigating virtualization of memory, per-process data structures like page tables show which pages and page frames can be accessed by a process at any given time. Any form of data created and managed by the operating system that keeps track of such access decisions for future reference is often called a **credential**.

If the operating system has not already produced a credential showing that an agent process can access a particular object, however, it needs information about the identity of the process's principal to determine if its request should be granted. Different operating systems have used different types of identity for principals. For instance, most operating systems have a notion of a user identity, where the user is, typically, some human being. (The concept of a user has been expanded over the years to increase its power, as we'll see later.) So perhaps all processes run by a particular person will have the same identity associated with them. Another common type of identity is a group of users. In a manufacturing company, you might want to give all your salespersons access to your inventory information, so they can determine how many widgets and whizz-bangs you have in the warehouse, while it wouldn't be necessary for your human resources personnel to have access to that information[2]. Yet another form of identity is the program that the process is running. Recall that a process is a running version of a program. In some systems (such as the Android Operating System), you can grant certain privileges to particular programs. Whenever they run, they can use these privileges, but other programs cannot.

Regardless of the kind of identity we use to make our security decisions, we must have some way of attaching that identity to a particular process. Clearly, this attachment is a crucial security issue. If you

[1] Another computer science overloading of the word "object." Here, it does not refer to "object oriented," but to the more general concept of a specific resource with boundaries and behaviors, such as a file or an IPC channel.

[2] Remember the principle of least privilege from the previous chapter? Here's an example of using it. A rogue human services employee won't be able to order your warehouse emptied of pop-doodles if you haven't given such employees the right to do so. As you read through the security chapters of this book, keep your eyes out for other applications of the security principles we discussed earlier.

misidentify a programmer employee process as an accounting department employee process, you could end up with an empty bank account. (Not to mention needing to hire a new programmer.) Or if you fail to identify your company president correctly when he or she is trying to give an important presentation to investors, you may find yourself out of a job once the company determines that you're the one who derailed the next round of startup capital, because the system didn't allow the president to access the presentation that would have bowled over some potential investors.

On the other hand, since everything except the operating system's own activities are performed by some process, if we can get this right for processes, we can be pretty sure we will have the opportunity to check our policy on every important action. But we need to bear in mind one other important characteristic of operating systems' usual approach to authentication: once a principal has been authenticated, systems will almost always rely on that authentication decision for at least the lifetime of the process. This characteristic puts a high premium on getting it right. Mistakes won't be readily corrected. Which leads to the crux:

> CRUX: HOW TO SECURELY IDENTIFY PROCESSES
>
> For systems that support processes belonging to multiple principals, how can we be sure that each process has the correct identity attached? As new processes are created, how can we be sure the new process has the correct identity? How can we be sure that malicious entities cannot improperly change the identity of a process?

54.2 Attaching Identities To Processes

Where do processes come from? Usually they are created by other processes. One simple way to attach an identity to a new process, then, is to copy the identity of the process that created it. The child inherits the parent's identity. Mechanically, when the operating system services a call from old process A to create new process B (`fork`, for example), it consults A's process control block to determine A's identity, creates a new process control block for B, and copies in A's identity. Simple, no?

That's all well and good if all processes always have the same identity. We can create a primal process when our operating system boots, perhaps assigning it some special system identity not assigned to any human user. All other processes are its descendants and all of them inherit that single identity. But if there really is only one identity, we're not going to be able to implement any policy that differentiates the privileges of one process versus another.

We must arrange that some processes have different identities and use those differences to manage our security policies. Consider a multi-user system. We can assign identities to processes based on which human user they belong to. If our security policies are primarily about some people being allowed to do some things and others not being allowed to, we now have an idea of how we can go about making our decisions.

If processes have a security-relevant identity, like a user ID, we're going to have to set the proper user ID for a new process. In most systems, a user has a process that he or she works with ordinarily: the shell process in command line systems, the window manager process in window-oriented system – you had figured out that both of these had to be processes themselves, right? So when you type a command into a shell or double click on an icon to start a process in a windowing system, you are asking the operating system to start a new process under your identity.

Great! But we do have another issue to deal with. How did that shell or window manager get your identity attached to itself? Here's where a little operating system privilege comes in handy. When a user first starts interacting with a system, the operating system can start a process up for that user. Since the operating system can fiddle with its own data structures, like the process control block, it can set the new process's ownership to the user who just joined the system.

Again, well and good, but how did the operating system determine the user's identity so it could set process ownership properly? You probably can guess the answer - the user logged in, implying that the user provided identity information to the OS proving who the user was. We've now identified a new requirement for the operating system: it must be able to query identity from human users and verify that they are who they claim to be, so we can attach reliable identities to processes, so we can use those identities to implement our security policies. One thing tends to lead to another in operating systems.

So how does the OS do that? As should be clear, we're building a towering security structure with unforeseeable implications based on the OS making the right decision here, so it's important. What are our options?

54.3 How To Authenticate Users?

So this human being walks up to a computer...

Assuming we leave aside the possibilities for jokes, what can be done to allow the system to determine who this person is, with reasonable accuracy? First, if the person is not an authorized user of the system at all, we should totally reject this attempt to sneak in. Second, if he or she is an authorized user, we need to determine, which one?

Classically, authenticating the identity of human beings has worked in one of three ways:

- Authentication based on what you know
- Authentication based on what you have
- Authentication based on what you are

When we say "classically" here, we mean "classically" in the, well, classical sense. Classically as in going back to the ancient Greeks and

Romans. For example, Polybius, writing in the second century B.C., describes how the Roman army used "watchwords" to distinguish friends from foes [P-46], an example of authentication based on what you know. A Roman architect named Celer wrote a letter of recommendation (which still survives) for one of his slaves to be given to an imperial procurator at some time in the 2nd century AD [C100] – authentication based on what the slave had. Even further back, in (literally) Biblical times, the Gileadites required refugees after a battle to say the word "shibboleth," since the enemies they sought (the Ephraimites) could not properly pronounce that word [JB-500]. This was a form of authentication by what you are: a native speaker of the Gileadites' dialect or of the Ephraimite dialect.

Having established the antiquity of these methods of authentication, let's leap past several centuries of history to the Computer Era to discuss how we use them in the context of computer authentication.

54.4 Authentication By What You Know

Authentication by what you know is most commonly performed by using passwords. Passwords have a long (and largely inglorious) history in computer security, going back at least to the CTSS system at MIT in the early 1960s [MT79]. A password is a secret known only to the party to be authenticated. By divulging the secret to the computer's operating system when attempting to log in, the party proves their identity. (You should be wondering about whether that implies that the system must also know the password, and what further implications that might have. We'll get to that.) The effectiveness of this form of authentication depends, obviously, on several factors. We're assuming other people don't know the party's password. If they do, the system gets fooled. We're assuming that no one else can guess it, either. And, of course, that the party in question must know (and remember) it.

Let's deal with the problem of other people knowing a password first. Leaving aside guessing, how could they know it? Someone who already knows it might let it slip, so the fewer parties who have to know it, the fewer parties we have to worry about. The person we're trying to authenticate has to know it, of course, since we're authenticating this person based on the person knowing it. We really don't want anyone else to be able to authenticate as that person to our system, so we'd prefer no third parties know the password. Thinking broadly about what a "third party" means here, that also implies the user shouldn't write the password down on a slip of paper, since anyone who steals the paper now knows the password. But there's one more party who would seem to need to know the password: our system itself. That suggests another possible vulnerability, since the system's copy of our password might leak out[3].

[3] "Might" is too weak a word. The first known incident of such stored passwords leaking is from 1962 [MT79]; such leaks happen to this day with depressing regularity and much larger scope. [KA16] discusses a leak of over 100 million passwords stored in usable form.

TIP: AVOID STORING SECRETS

Storing secrets like plaintext passwords or cryptographic keys is a hazardous business, since the secrets usually leak out. Protect your system by not storing them if you don't need to. If you do need to, store them in a hashed form using a strong cryptographic hash. If you can't do that, encrypt them with a secure cipher. (Perhaps you're complaining to yourself that we haven't told you about those yet. Be patient.) Store them in as few places, with as few copies, as possible. Don't forget temporary editor files, backups, logs, and the like, since the secrets may be there, too. Remember that anything you embed into an executable you give to others will not remain secret, so it's particularly dangerous to store secrets in executables. In some cases, even secrets only kept in the heap of an executing program have been divulged, so avoid storing and keeping secrets even in running programs.

Interestingly enough, though, our system does not actually need to know the password. Think carefully about what the system is doing when it checks the password the user provides. It's checking to see if the user knows it, not what that password actually is. So if the user provides us the password, but we don't know the password, how on earth could our system do that?

You already know the answer, or at least you'll slap your forehead and say "I should have thought of that" once you hear it. Store a **hash** of the password, not the password itself. When the user provides you with what he or she claims to be the password, hash the claim and compare it to the stored hashed value. If it matches, you believe he or she knows the password. If it doesn't, you don't. Simple, no? And now your system doesn't need to store the actual password. That means if you're not too careful with how you store the authentication information, you haven't actually lost the passwords, just their hashes. By their nature, you can't reverse hashing algorithms, so the adversary can't use the stolen hash to obtain the password. If the attacker provides the hash, instead of the password, the hash itself gets hashed by the system, and a hash of a hash won't match the hash.

There is a little more to it than that. The benefit we're getting by storing a hash of the password is that if the stored copy is leaked to an attacker, the attacker doesn't know the passwords themselves. But it's not quite enough just to store something different from the password. We also want to ensure that whatever we store offers an attacker no help in guessing what the password is. If an attacker steals the hashed password, he or she should not be able to analyze the hash to get any clues about the password itself. There is a special class of hashing algorithms called **cryptographic hashes** that make it infeasible to use the hash to figure out what the password is, other than by actually passing a guess at the password through the hashing algorithm. One unfortunate characteristic of cryptographic hashes is that they're hard to design, so even smart people shouldn't try. They use ones created by experts. That's what modern systems should do with password hashing: use a cryptographic hash that has been thoroughly studied and has no known flaws. At any given time, which cryptographic hashing algorithms meet those requirements may vary. At the time of this writing, SHA-3 [B+09] is the US standard for cryptographic hash algorithms, and is a good choice.

Let's move on to the other problem: guessing. Can an attacker who wants to pose as a user simply guess the password? Consider the simplest possible password: a single bit, valued 0 or 1. If your password is a single bit long, then an attacker can try guessing "0" and have a 50/50 chances of being right. Even if wrong, if a second guess is allowed, the attacker now knows that the password is "1" and will correctly guess that.

Obviously, a one bit password is too easy to guess. How about an 8 bit password? Now there are 256 possible passwords you could choose. If the attacker guesses 256 times, sooner or later the guess will be right, taking 128 guesses (on average). Better than only having to guess twice, but still not good enough. It should be clear to you, at this point, that the length of the password is critical in being resistant to guessing. The longer the password, the harder to guess.

But there's another important factor, since we normally expect human beings to type in their passwords from keyboards or something similar. And given that we've already ruled out writing the password down somewhere as insecure, the person has to remember it. Early uses of passwords addressed this issue by restricting passwords to letters of the alphabet. While this made them easier to type and remember, it also cut down heavily on the number of bit patterns an attacker needed to guess to find someone's password, since all of the bit patterns that did not represent alphabetic characters would not appear in passwords. Over time, password systems have tended to expand the possible characters in a password, including upper and lower case letters, numbers, and special characters. The more possibilities, the harder to guess.

So we want long passwords composed of many different types of characters. But attackers know that people don't choose random strings of these types of characters as their passwords. They often choose names or familiar words, because those are easy to remember. Attackers trying to guess passwords will thus try lists of names and words before trying random strings of characters. This form of password guessing is called a **dictionary attack**, and it can be highly effective. The dictionary here isn't Websters (or even the Oxford English Dictionary), but rather is a specialized list of words, names, meaningful strings of numbers (like "123456"), and other character patterns people tend to use for passwords, ordered by the probability that they will be chosen as the password. A good dictionary attack can figure out 90% of the passwords for a typical site [G13].

If you're smart in setting up your system, an attacker really should not be able to run a dictionary attack on a login process remotely. With any care at all, the attacker will not guess a user's password in the first five or

ASIDE: PASSWORD VAULTS

One way you can avoid the problem of choosing passwords is to use what's called a password vault or key chain. This is an encrypted file kept on your computer that stores passwords. It's encrypted with a password of its own. To get passwords out of the vault, you must provide the password for the vault, reducing the problem of remembering a different password for every site to remembering one password. Also, it ensures that attackers can only use your passwords if they not only have the special password that opens the vault, but they have access to the vault itself. Of course, the benefits of securely storing passwords this way are limited to the strength of the passwords stored in the vault, since guessing and dictionary attacks will still work. Some password vaults will generate strong passwords for you – not very memorable ones, but that doesn't matter, since it's the vault that needs to remember it, not you. You can also find password vaults that store your passwords in the cloud. If you provide them with cleartext versions of your password to store them, however, you are sharing a password with another entity that doesn't really need to know it, thus taking a risk that perhaps you shouldn't take. If the cloud stores only your encrypted passwords, the risk is much lower.

six guesses (alas, sometimes no care is taken and the attacker will), and there's no good reason your system should allow a remote user to make 15,000 guesses at an account's password without getting it right. So by either shutting off access to an account when too many wrong guesses are made at its password, or (better) by drastically slowing down the process of password checking after a few wrong guesses (which makes a long dictionary attack take an infeasible amount of time), you can protect the account against such attacks.

But what if the attacker stole your password file? Since we assume you've been paying attention, it contains hashes of passwords, not passwords itself. But we also assume you paid attention when we told you to use a widely known cryptographic hash, and if you know it, so does the person who stole your password file. If the attacker obtained your hashed passwords, the hashing algorithm, a dictionary, and some compute power, the attacker can crank away at guessing your passwords at their leisure. Worse, if everyone used the same cryptographic hashing algorithm (which, in practice, they probably will), the attacker only needs to run each possible password through the hash once and store the results (essentially, the dictionary has been translated into hashed form). So when the attacker steals your password file, he or she would just need to do string comparisons to your hashed passwords and the newly created dictionary of hashed passwords, which is much faster.

There's a simple fix: before hashing a new password and storing it in your password file, generate a big random number (say 32 or 64 bits) and concatenate it to the password. Hash the result and store that. You also need to store that random number, since when the user tries to log in and provides the correct password, you'll need to take what the user provided, concatenate the stored random number, and run that through the hashing algorithm. Otherwise, the password hashed by itself won't match what you stored. You typically store the random number (which is called a **salt**) in the password file right next to the hashed password. This concept was introduced in Robert Morris and Ken Thompson's early paper on password security [MT79].

Why does this help? The attacker can no longer create one translation of passwords in the dictionary to their hashes. What is needed is one translation for every possible salt, since the password files that were stolen are likely to have a different salt for every password. If the salt is 32 bits, that's 2^{32} different translations for each word in the dictionary, which makes the approach of pre-computing the translations infeasible. Instead, for each entry in the stolen password file, the dictionary attack must freshly hash each guess with the password's salt. The attack is still feasible if you have chosen passwords badly, but it's not nearly as cheap. Any good system that uses passwords and cares about security stores cryptographically hashed and salted passwords. If yours doesn't, you're putting your users at risk.

There are other troubling issues for the use of passwords, but many of those are not particular to the OS, so we won't fling further mud at them here. Suffice it to say that there is a widely held belief in the computer security community that passwords are a technology of the past, and are no longer sufficiently secure for today's environments. At best, they can serve as one of several authentication mechanisms used in concert. This idea is called **multi-factor authentication**, with **two-factor authentication** being the version that gets the most publicity. You're perhaps already familiar with the concept: to get money out of an ATM, you need to know your personal identification number (PIN). That's essentially a password. But you also need to provide further evidence of your identity...

54.5 Authentication by What You Have

Most of us have probably been in some situation where we had an identity card that we needed to show to get us into somewhere. At least, we've probably all attended some event where admission depended on having a ticket for the event. Those are both examples of authentication based on what you have, an ID card or a ticket, in these cases.

When authenticating yourself to an operating system, things are a bit different. In special cases, like the ATM mentioned above, the device (which has, after all, a computer inside – you knew that, right?) has special hardware to read our ATM card. That hardware allows it to determine that, yes, we have that card, thus providing the further proof to go along with your PIN. Most desktop computers, laptops, tablets, smart phones, and the like do not have that special hardware. So how can they tell what we have?

ASIDE: LINUX LOGIN PROCEDURES

Linux, in the tradition of earlier Unix systems, authenticates users based on passwords and then ties that identity to an initial process associated with the newly logged in user, much as described above. Here we will provide a more detailed step-by-step description of what actually goes on when a user steps up to a keyboard and tries to log in to a Unix system, as a solid example of how a real operating system handles this vital security issue.

1. A special login process running under a privileged system identity displays a prompt asking for the user to type in his or her identity, in the form of a generally short user name. The user types in a user name and hits carriage return. The name is echoed to the terminal.
2. The login process prompts for the user's password. The user types in the password, which is not echoed.
3. The login process looks up the name the user provided in the password file. If it is not found, the login process rejects the login attempt. If it is found, the login process determines the internal user identifier (a unique user ID number), the group (another unique ID number) that the user belongs to, the initial command shell that should be provided to this user once login is complete, and the home directory that shell should be started in. Also, the login process finds the salt and the salted, hashed version of the correct password for this user, which are permanently stored in a secure place in the system.
4. The login process combines the salt for the user's password and the password provided by the user and performs the hash on the combination. It compares the result to the stored version obtained in the previous step. If they do not match, the login process rejects the login attempt.
5. If they do match, fork a process. Set the user and group of the forked process to the values determined earlier, which the privileged identity of the login process is permitted to do. Change directory to the user's home directory and exec the shell process associated with this user (both the directory name and the type of shell were determined in step 3).

There are some other details associated with ensuring that we can log in another user on the same terminal after this one logs out that we don't go into here.

Note that in steps 3 and 4, login can fail either because the user name is not present in the system or because the password does not match the user name. Linux and most other systems do not indicate which condition failed, if one of them did. This choice prevents attackers from learning the names of legitimate users of the system just by typing in guesses, since they cannot know if they guessed a non-existent name or guessed the wrong password for a legitimate user name. Not providing useful information to non-authenticated users is generally a good security idea that has applicability in other types of systems.

Think a bit about why Linux's login procedure chooses to echo the typed user name when it doesn't echo the password. Is there no security disadvantage to echoing the user name, is it absolutely necessary to echo the user name, or is it a tradeoff of security for convenience? Why not echo the password?

If we have something that plugs into one of the ports on a computer, such as a hardware token that uses USB, then, with suitable software support, the operating system can tell whether the user trying to log in has the proper device or not. Some security tokens (sometimes called **dongles**, an unfortunate choice of name) are designed to work that way.

In other cases, since we're trying to authenticate a human user anyway, we make use of the person's capabilities to transfer information from whatever it is he or she has to the system where the authentication is required. For example, some smart tokens display a number or character string on a tiny built-in screen. The human user types the information read off that screen into the computer's keyboard. The operating system does not get direct proof that the user has the device, but if only someone with access to the device could know what information was supposed to be typed in, the evidence is nearly as good.

These kinds of devices rely on frequent changes of whatever information the device passes (directly or indirectly) to the operating system, perhaps every few seconds, perhaps every time the user tries to authenticate himself or herself. Why? Well, if it doesn't, anyone who can learn the static information from the device no longer needs the device to pose as the user. The authentication mechanism has been converted from "something you have" to "something you know," and its security now depends on how hard it is for an attacker to learn that secret.

One weak point for all forms of authentication based on what you have is, what if you don't have it? What if you left your smartphone on your dresser bureau this morning? What if your dongle slipped out of your pocket on your commute to work? What if a subtle pickpocket brushed up against you at the coffee shop and made off with your secret authentication device? You now have a two-fold problem. First, you don't have the magic item you need to authenticate yourself to the operating system. You can whine at your computer all you want, but it won't care. It will continue to insist that you produce the magic item you lost. Second, someone else has your magic item, and possibly they can pretend to be you, fooling the operating system that was relying on authentication by what you have. Note that the multi-factor authentication we mentioned earlier can save your bacon here, too. If the thief stole your security token, but doesn't know your password, the thief will still have to guess that before they can pose as you[4].

If you study system security in practice for very long, you'll find that there's a significant gap between what academics (like me) tell you is safe and what happens in the real world. Part of this gap is because the real world needs to deal with real issues, like user convenience. Part of it is because security academics have a tendency to denigrate anything where they can think of a way to subvert it, even if that way is not itself particularly practical. One example in the realm of authentication mechanisms

[4] Assuming, of course, you haven't written the password with a Sharpie onto the back of the smart card the thief stole. Well, it seemed like a good idea at the time...

based on what you have is authenticating a user to a system by sending a text message to the user's cell phone. The user then types a message into the computer. Thinking about this in theory, it sounds very weak. In addition to the danger of losing the phone, security experts like to think about exotic attacks where the text message is misdirected to the attacker's phone, allowing the attacker to provide the secret information from the text message to the computer.

In practice, people usually have their phone with them and take reasonable care not to lose it. If they do lose it, they notice that quickly and take equally quick action to fix their problem. So there is likely to be a relatively small window of time between when your phone is lost and when systems learn that they can't authenticate you using that phone. Also in practice, redirecting text messages sent to cell phones is possible, but far from trivial. The effort involved is likely to outweigh any benefit the attacker would get from fooling the authentication system, at least in the vast majority of cases. So a mechanism that causes security purists to avert their gazes in horror in actual use provides quite reasonable security[5]. Keep this lesson in mind. Even if it isn't on the test[6], it may come in handy some time in your later career.

54.6 Authentication by What You Are

If you don't like methods like passwords and you don't like having to hand out smart cards or security tokens to your users, there is another option. Human beings (who are what we're talking about authenticating here) are unique creatures with physical characteristics that differ from all others, sometimes in subtle ways, sometimes in obvious ones. In addition to properties of the human body (from DNA at the base up to the appearance of our face at the top), there are characteristics of human behavior that are unique, or at least not shared by very many others. This observation suggests that if our operating system can only accurately measure these properties or characteristics, it can distinguish one person from another, solving our authentication problem.

This approach is very attractive to many people, most especially to those who have never tried to make it work. Going from the basic observation to a working, reliable authentication system is far from easy. But it can be made to work, to much the same extent as the other authentication mechanisms. We can use it, but it won't be perfect, and has its own set of problems and challenges.

[5]However, in 2016 the United States National Institute of Standards and Technology issued draft guidance deprecating the use of this technique for two-factor authentication, at least in some circumstances. Here's another security lesson: what works today might not work tomorrow.

[6]We don't know about you, but every time the word "test" or "quiz" or "exam" comes up, our heart skips a beat or two. Too many years of being a student will do this to a person.

Remember that we're talking about a computer program (either the OS itself or some separate program it invokes for the purpose) measuring a human characteristic and determining if it belongs to a particular person. Think about what that entails. What if we plan to use facial recognition with the camera on a smart phone to authenticate the owner of the phone? If we decide it's the right person, we allow whoever we took the picture of to use the phone. If not, we give them the raspberry (in the cyber sense) and keep them out.

You should have identified a few challenges here. First, the camera is going to take a picture of someone who is, presumably, holding the phone. Maybe it's the owner, maybe it isn't. That's the point of taking the picture. If it isn't, we should assume whoever it is would like to fool us into thinking that they are the actual owner. What if it's someone who looks a lot like the right user, but isn't? What if the person is wearing a mask? What if the person holds up a photo of the right user, instead of their own face? What if the lighting is dim, or the person isn't fully facing the camera? Alternately, what if it is the right user and the person is not facing the camera, or the lighting is dim, or something else has changed about the person's look? (e.g., hairstyle)

Computer programs don't recognize faces the way people do. They do what programs always do with data: they convert it to zeros and ones and process it using some algorithm. So that "photo" you took is actually a collection of numbers, indicating shadow and light, shades of color, contrasts, and the like. OK, now what? Time to decide if it's the right person's photo or not! How?

If it were a password, we could have stored the right password (or, better, a hash of the right password) and done a comparison of what got typed in (or its hash) to what we stored. If it's a perfect match, authenticate. Otherwise, don't. Can we do the same with this collection of zeros and ones that represent the picture we just took? Can we have a picture of the right user stored permanently in some file (also in the form of zeros and ones) and compare the data from the camera to that file?

Probably not in the same way we compared the passwords. Consider one of those factors we just mentioned above: lighting. If the picture we stored in the file was taken under bright lights and the picture coming out of the camera was taken under dim lights, the two sets of zeros and ones are most certainly not going to match. In fact, it's quite unlikely that two pictures of the same person, taken a second apart under identical conditions, would be represented by exactly the same set of bits. So clearly we can't do a comparison based on bit-for-bit equivalence.

Instead, we need to compare based on a higher-level analysis of the two photos, the stored one of the right user and the just-taken one of the person who claims to be that user. Generally this will involve extracting higher-level features from the photos and comparing those. We might, for example, try to calculate the length of the nose, or determine the color of the eyes, or make some kind of model of the shape of the mouth. Then we would compare the same feature set from the two photos.

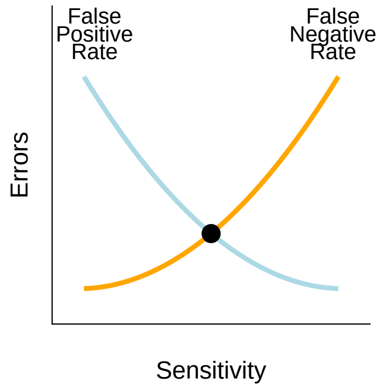


Figure 54.1: **Crossover Error Rate**

Even here, though, an exact match is not too likely. The lighting, for example, might slightly alter the perceived eye color. So we'll need to allow some sloppiness in our comparison. If the feature match is "close enough," we authenticate. If not, we don't. We will look for close matches, not perfect matches, which brings the nose of the camel of tolerances into our authentication tent. If we are intolerant of all but the closest matches, on some days we will fail to match the real user's picture to the stored version. That's called a **false negative**, since we incorrectly decided not to authenticate. If we are too tolerant of differences in measured versus stored data, we will authenticate a user whom is not who they claim to be. That's a **false positive**, since we incorrectly decided to authenticate.

The nature of biometrics is that any implementation will have a characteristic false positive and false negative rate. Both are bad, so you'd like both to be low. For any given implementation of some biometric authentication technique, you can typically tune it to achieve some false positive rate, or tune it to achieve some false negative rate. But you usually can't minimize both. As the false positive rate goes down, the false negative rate goes up, and vice versa. The **sensitivity** describes how close the match must be.

Figure 54.1 shows the typical relationship between these error rates. Note the circle at the point where the two curves cross. That point represents the crossover error rate, a common metric for describing the accuracy of a biometric. It represents an equal tradeoff between the two kinds of errors. It's not always the case that one tunes a biometric system to hit the crossover error rate, since you might care more about one kind of error than the other. For example, a smart phone that frequently locks its legitimate user out because it doesn't like today's fingerprint reading is not going to be popular, while the chances of a thief who stole the phone having a similar fingerprint are low. Perhaps low false negatives matter more here. On the other hand, if you're opening a bank vault with a retinal scan, requiring the bank manager to occasionally provide a second scan isn't too bad, while allowing a robber to open the vault with a bogus fake eye would be a disaster. Low false positives might be better here.

Leaving aside the issues of reliability of authentication using biometrics, another big issue for using human characteristics to authenticate is that many of the techniques for measuring them require special hardware not likely to be present on most machines. Many computers (including smart phones, tablets, and laptops) are likely to have cameras, but embedded devices and server machines probably don't. Relatively few machines have fingerprint readers, and even fewer are able to measure more exotic biometrics. While a few biometric techniques (such as measuring typing patterns) require relatively common hardware that is likely to be present on many machines anyway, there aren't many such techniques. Even if a special hardware device is available, the convenience of using them for this purpose can be limiting.

One further issue you want to think about when considering using biometric authentication is whether there is any physical gap between where the biometric quantity is measured and where it is checked. In particular, checking biometric readings provided by an untrusted machine across the network is hazardous. What comes in across the network is simply a pattern of bits spread across one or more messages, whether it represents a piece of a web page, a phoneme in a VoIP conversation, or part of a scanned fingerprint. Bits are bits, and anyone can create any bit pattern they want. If a remote adversary knows what the bit pattern representing your fingerprint looks like, they may not need your finger, or even a fingerprint scanner, to create it and feed it to your machine. When the hardware performing the scanning is physically attached to your machine, there is less opportunity to slip in a spurious bit pattern that didn't come from the device. When the hardware is on the other side of the world on a machine you have no control over, there is a lot more opportunity. The point here is to be careful with biometric authentication information provided to you remotely.

In all, it sort of sounds like biometrics are pretty terrible for authentication, but that's the wrong lesson. For that matter, previous sections probably made it sound like all methods of authentication are terrible. Certainly none of them are perfect, but your task as a system designer is not to find the perfect authentication mechanism, but to use mechanisms that are well suited to your system and its environment. A good fingerprint reader built in to a smart phone might do its job quite well. A long, unguessable password can provide a decent amount of security. Well-designed smart cards can make it nearly impossible to authenticate yourself without having them in your hand. And where each type of mechanism fails, you can perhaps correct for that failure by using a second or third authentication mechanism that doesn't fail in the same cases.

54.7 Authenticating Non-Humans

No, we're not talking about aliens or extra-dimensional beings, or even your cat. If you think broadly about how computers are used today, you'll see that there are many circumstances in which no human user is associated with a process that's running. Consider a web server. There really isn't some human user logged in whose identity should be attached to the web server. Or think about embedded devices, such as a smart light bulb. Nobody logs in to a light bulb, but there is certainly code running there, and quite likely it is process-oriented code.

Mechanically, the operating system need not have a problem with the identities of such processes. Simply set up a user called `webserver` or `lightbulb` on the system in question and attach the identity of that "user" to the processes that are associated with running the web server or turning the light bulb on and off. But that does lead to the question of how you make sure that only real web server processes are tagged with that identity. We wouldn't want some arbitrary user on the web server machine creating processes that appear to belong to the server, rather than to that user.

One approach is to use passwords for these non-human users, as well. Simply assign a password to the web server user. When does it get used? When it's needed, which is when you want to create a process belonging to the web server, but you don't already have one in existence. The system administrator could log in as the web server user, creating a command shell and using it to generate the actual processes the server needs to do its business. As usual, the processes created by this shell process would inherit their parent's identity, `webserver`, in this case. More commonly, we skip the go-between (here, the login) and provide some mechanism whereby the privileged user is permitted to create processes that belongs not to that user, but to some other user such as `webserver`. Alternately, we can provide a mechanism that allows a process to change its ownership, so the web server processes would start off under some other user's identity (such as the system administrator's) and change their ownership to `webserver`. Yet another approach is to allow a temporary change of process identity, while still remembering the original identity. (We'll say more about this last approach in a future chapter.) Obviously, any of these approaches require strong controls, since they allow one user to create processes belonging to another user.

As mentioned above, passwords are the most common authentication method used to determine if a process can be assigned to one of these non-human users. Sometimes no authentication of the non-human user is required at all, though. Instead, certain other users (like trusted system administrators) are given the right to assign new identities to the processes they create, without providing any further authentication information than their own. In Linux and other Unix systems, the sudo command offers this capability. For example, if you type the following:

```
sudo -u webserver apache2
```

ASIDE: OTHER AUTHENTICATION POSSIBILITIES

Usually, what you know, what you have, and what you are cover the useful authentication possibilities, but sometimes there are other options. Consider going into the Department of Motor Vehicles to apply for a driver's license. You probably go up to a counter and talk to some employee behind that counter, perhaps giving the person a bunch of personal information, maybe even money to cover a fee for the license. Why on earth did you believe that person was actually a DMV employee who was able to get you a legitimate driver's license? You probably didn't know the person; you weren't shown an official ID card; the person didn't recite the secret DMV mantra that proved he or she was an initiate of that agency. You believed it because the person was standing behind a particular counter, which is the counter DMV employees stand behind. You authenticated the person based on location.

Once in a while, that approach can be handy in computer systems, most frequently in mobile or pervasive computing. If you're tempted to use it, think carefully about how you're obtaining the evidence that the subject really is in a particular place. It's actually fairly tricky.

What else? Perhaps you can sometimes authenticate based on what someone does. If you're looking for personally characteristic behavior, like their typing pattern or delays between commands, that's a type of biometric. (Google introduced multi-factor authentication of this kind in its Android phones, for example.) But you might be less interested in authenticating exactly who they are versus authenticating that they belong to the set of Well Behaved Users. Many web sites, for example, care less about who their visitors are and more about whether they use the web site properly. In this case, you might authenticate their membership in the set by their ongoing interactions with your system.

This would indicate that the apache2 program should be started under the identity of `webserver`, rather than under the identity of whoever ran the sudo command. This command might require the user running it to provide their own authentication credentials (for extra certainty that it really is the privileged user asking for it, and not some random visitor accessing the computer during the privileged user's coffee break), but would not require authentication information associated with `webserver`. Any sub-processes created by apache2 would, of course, inherit the identity of `webserver`. We'll say more about sudo in the chapter on access control.

One final identity issue we alluded to earlier is that sometimes we wish to identify not just individual users, but groups of users who share common characteristics, usually security-related characteristics. For example, we might have four or five system administrators, any one of whom is allowed to start up the web server. Instead of associating the privilege with each one individually, it's advantageous to create a system-meaningful group of users with that privilege. We would then indicate that the four or five administrators are members of that group. This kind of group is another example of a security-relevant principal, since we will make our decisions on the basis of group membership, rather than individual identity. When one of the system administrators wished to do something requiring group membership, we would check that he or she was a member. We can either associate a group membership with each process, or use the process's individual identity information as an index into a list of groups that people belong to. The latter is more flexible, since it allows us to put each user into an arbitrary number of groups.

Most modern operating systems, including Linux and Windows, support these kinds of groups, since they provide ease and flexibility in dealing with application of security policies. They handle group membership and group privileges in manners largely analogous to those for individuals. For example, a child process will usually have the same group-related privileges as its parent. When working with such systems, it's important to remember that group membership provides a second path by which a user can obtain access to a resource, which has its benefits and its dangers.

54.8 Summary

If we want to apply security policies to actions taken by processes in our system, we need to know the identity of the processes, so we can make proper decisions. We start the entire chain of processes by creating a process at boot time belonging to some system user whose purpose is to authenticate users. They log in, providing authentication information in one or more forms to prove their identity. The system verifies their identity using this information and assigns their identity to a new process that allows the user to go about their business, which typically involves running other processes. Those other processes will inherit the user's identity from their parent process. Special secure mechanisms can allow identities of processes to be changed or to be set to something other than the parent's identity. The system can then be sure that processes belong to the proper user and can make security decisions accordingly.

Historically and practically, the authentication information provided to the system is either something the authenticating user knows (like a password or PIN), something the user has (like a smart card or proof of possession of a smart phone), or something the user is (like the user's fingerprint or voice scan). Each of these approaches has its strengths and weaknesses. A higher degree of security can be obtained by using multi-factor authentication, which requires a user to provide evidence of more than one form, such as requiring both a password and a one-time code that was texted to the user's smart phone.

References

[B+09] "The road from Panama to Keccak via RadioGatun" by Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche. *The authors who developed SHA-3. For a more readable version, try the Wikipedia page first about SHA-3. There, you learn about the "sponge construction", which actually has something to do with cryptographic hashes, and not the cleaning of your kitchen.*

[C100] "Letter of recommendation to Tiberius Claudius Hermeros" by Celer the Architect. Circa 100 A.D.. *This letter introduced a slave to the imperial procurator, thus providing said procurator evidence that the slave was who he claimed to be. Read the translation at the following website* `http://papyri.info/ddbdp/c.ep.lat;;81`.

[G13] "Anatomy of a hack: even your 'complicated' password is easy to crack" by Dan Goodin. `http://www.wired.co.uk/article/password-cracking`, May 2013. *A description of how three experts used dictionary attacks to guess a large number of real passwords, with 90% success.*

[JB-500] "Judges 12, verses 5-6" The Bible, roughly 5th century BC. *An early example of the use of biometrics. Failing this authentication had severe consequences, as the Gileadites slew mispronouncers, some 42,000 of them according to the book of Judges.*

[KA16] *VK.com Hacked! 100 Million Clear Text Passwords Leaked Online* by Swati Khandelwal. `http://thehackernews.com/2016/06/vk-com-data-breach.html`. *One of many reports of stolen passwords stored in plaintext form.*

[MT79] "Password Security: A Case History" by Robert Morris and Ken Thompson. Communications of the ACM, Vol. 22, No. 11, 1979. *A description of the use of passwords in early Unix systems. It also talks about password shortcomings from more than a decade earlier, in the CTSS system. And it was the first paper to discuss the technique of password salting.*

[M+02] "Impact of Artificial "Gummy" Fingers on Fingerprint Systems" by Tsutomu Matsumoto, Hiroyuki Matsumoto, Koji Yamada, and Satoshi Hoshino. SPIE Vol. #4677, January 2002. *A neat example of how simple ingenuity can reveal the security weaknesses of systems. In this case, the researchers showed how easy it was to fool commercial fingerprint reading machines.*

[P-46] "The Histories" by Polybius. Circa 146 B.C.. *A history of the Roman Republic up to 146 B.C. Polybius provides a reasonable amount of detail not only about how the Roman Army used watchwords to authenticate themselves, but how they distributed them where they needed to be, which is still a critical element of using passwords.*

[TR78] "On the Extraordinary: An Attempt at Clarification" by Marcello Truzzi. Zetetic Scholar, Vol. 1, No. 1, p. 11, 1978. *Truzzi was a scholar who investigated various pseudoscience and paranormal claims. He is unusual in this company in that he insisted that one must actually investigate such claims before dismissing them, not merely assume they are false because they conflict with scientific orthodoxy.*

55

Access Control

Chapter by **Peter Reiher (UCLA)**

55.1 Introduction

So we know what our security goals are, we have at least a general sense of the security policies we'd like to enforce, and we have some evidence about who is requesting various system services that might (or might not) violate our policies. Now we need to take that information and turn it into something actionable, something that a piece of software can perform for us.

There are two important steps here:

1. Figure out if the request fits within our security policy.
2. If it does, perform the operation. If not, make sure it isn't done.

The first step is generally referred to as **access control**. We will determine which system resources or services can be accessed by which parties in which ways under which circumstances. Basically, it boils down to another of those binary decisions that fit so well into our computing paradigms: yes or no. But how to make that decision? To make the problem more concrete, consider this case. User X wishes to read and write file /var/foo. Under the covers, this case probably implies that a process being run under the identity of User X issued a system call such as:

open("/var/foo", O_RDWR)

Note here that we're not talking about the Linux `open()` call, which is a specific implementation that handles access control a specific way. We're talking about the general idea of how you might be able to control access to a file open system call. Hence the different font, to remind you.

How should the system handle this request from the process, making sure that the file is not opened if the security policy to be enforced forbids it, but equally making sure that the file is opened if the policy allows it? We know that the system call will trap to the operating system, giving it the opportunity to do something to make this decision. Mechanically speaking, what should that "something" be?

THE CRUX OF THE PROBLEM:
HOW TO DETERMINE IF AN ACCESS REQUEST SHOULD BE GRANTED?

How can the operating system decide if a particular request made by a particular process belonging to a particular user at some given moment should or should not be granted? What information will be used to make this decision? How can we set this information to encode the security policies we want to enforce for our system?

55.2 Important Aspects Of The Access Control Problem

As usual, the system will run some kind of algorithm to make this decision. It will take certain inputs and produce a binary output, a yes-or-no decision on granting access. At the high level, access control is usually spoken of in terms of **subjects**, **objects**, and **access**. A subject is the entity that wants to perform the access, perhaps a user or a process. An object is the thing the subject wants to access, perhaps a file or a device. Access is some particular mode of dealing with the object, such as reading it or writing it. So an access control decision is about whether a particular subject is allowed to perform a particular mode of access on a particular object. We sometimes refer to the process of determining if a particular subject is allowed to perform a particular form of access on a particular[1] object as **authorization**.

One relevant issue is when will access control decisions be made? The system must run whatever algorithm it uses every time it makes such a decision. The code that implements this algorithm is called a **reference monitor**, and there is an obvious incentive to make sure it is implemented both correctly and efficiently. If it's not correct, you make the wrong access decisions – obviously bad. Its efficiency is important because it will inject some overhead whenever it is used. Perhaps we wish to minimize these overheads by not checking access control on every possible opportunity. On the other hand, remember that principle of complete mediation we introduced a couple of chapters back? That principle said we should check security conditions every time someone asked for something.

Clearly, we'll need to balance costs against security benefits. But if we can find some beneficial special cases where we can achieve low cost without compromising security, we can possibly manage to avoid trading off one for the other, at least in those cases.

One way to do so is to give subjects objects that belong only to them. If the object is inherently theirs, by its very nature and unchangeably so, the system can let the subject (a process, in the operating system case) ac-

[1]Wow. You know how hard it is to get so many instances of the word "particular" to line up like this? It's a column of particulars! But, perhaps, not particularly interesting.

cess it freely. Virtualization allows us to create virtual objects of this kind. Virtual memory is an excellent example. A process is allowed to access its virtual memory freely[2], with no special operating system access control check at the moment the process tries to use it. A good thing, too, since otherwise we would need to run our access control algorithm on every process memory reference, which would lead to a ridiculously slow system. We can play similar virtualization tricks with peripheral devices. If a process is given access to some virtual device, which is actually backed up by a real physical device controlled by the OS, and if no other process is allowed to use that device, the operating system need not check for access control every time the process wants to use it. For example, a process might be granted control of a GPU based on an initial access control decision, after which the process can write to the GPU's memory or issue instructions directly to it without further intervention by the OS.

Of course, as discussed earlier, virtualization is mostly an operating-system provided illusion. Processes share memory, devices, and other computing resources. What appears to be theirs alone is actually shared, with the operating system running around behind the scenes to keep the illusion going, sometimes assisted by special hardware. That means the operating system, without the direct knowledge and participation of the applications using the virtualized resource, still has to make sure that only proper forms of access to it are allowed. So merely relying on virtualization to ensure proper access just pushes the problem down to protecting the virtualization functionality of the OS. Even if we leave that issue aside, sooner or later we have to move past cheap special cases and deal with the general problem. Subject X wants to read and write object `/tmp/foo`. Maybe it's allowable, maybe it isn't. Now what?

Computer scientists have come up with two basic approaches to solving this question, relying on different data structures and different methods of making the decision. One is called **access control lists** and the other is called **capabilities**. It's actually a little inaccurate to claim that computer scientists came up with these approaches, since they've been in use in non-computer contexts for millennia. Let's look at them in a more general perspective before we consider operating system implementations.

Let's say we want to start an exclusive nightclub (called, perhaps, Chez Andrea[3]) restricted to only the best operating system researchers and developers. We don't want to let any of those database or programming language people slip in, so we'll need to make sure only our approved customers get through the door. How might we do that? One

[2]Almost. Remember the bits in the page table that determine whether a particular page can be read, written, or executed? But it's not the operating system doing the runtime check here, it's the virtual memory hardware.

[3]The authors Arpaci-Dusseau would like to note that author Reiher is in charge of these name choices for the security chapters, and did not strong-arm him into using their names throughout this and other examples. We now return you to your regular reading...

way would be to hire a massive intimidating bouncer who has a list of all the approved members. When someone wants to enter the club, they would prove their identity to the bouncer, and the bouncer would see if they were on the list. If it was Linus Torvalds or Barbara Liskov, the bouncer would let them in, but would keep out the hoi polloi networking folks who had failed to distinguish themselves in operating systems.

Another approach would be to put a really great lock on the door of the club and hand out keys to that lock to all of our OS buddies. If Jerome Saltzer wanted to get in to Chez Andrea, he'd merely pull out his key and unlock the door. If some computer architects with no OS chops wanted to get in, they wouldn't have a key and thus would be stuck outside. Compared to the other approach, we'd save on the salary of the bouncer, though we would have to pay for the locks and keys[4]. As new luminaries in the OS field emerge who we want to admit, we'll need new keys for them, and once in a while we may make a mistake and hand out a key to someone who doesn't deserve it, or a member might lose a key, in which case we need to make sure that key no longer opens the club door.

The same ideas can be used in computer systems. Early computer scientists decided to call the approach that's kind of like locks and keys a **capability-based system**, while the approach based on the bouncer and the list of those to admit was called an **access control list system**. Capabilities are thus like keys, or tickets to a movie, or tokens that let you ride a subway. Access control lists are thus like, well, lists. How does this work in an operating system? If you're using capabilities, when a process belonging to user X wants to read and write file `/tmp/foo`, it hands a capability specific to that file to the system. (And precisely what, you may ask, is a capability in this context? Good question! We'll get to that.) If you're using access control lists (**ACLs**, for short), the system looks up user X on an ACL associated with `/tmp/foo`, only allowing the access if the user is on the list. In either case, the check can be made at the moment the access (an `open()` call, in our example) is requested. The check is made after trapping to the operating system, but before the access is actually permitted, with an early exit and error code returned if the access control check fails.

At a high level, these two options may not sound very different, but when you start thinking about the algorithm you'll need to run and the data structures required to support that algorithm, you'll quickly see that there are major differences. Let's walk through each in turn.

[4]Note that for both access control lists and capabilities, we are assuming we've already authenticated the person trying to enter the club. If some nobody wearing a Linus Torvalds or Barbara Liskov mask gets past our bouncer, or if we aren't careful to determine that it really is Jerome Saltzer before handing a random person the key, we're not going to keep the riffraff out. Abandoning the cute analogy, absolutely the same issue applies in real computer systems, which is why the previous chapter discussed authentication in detail.

55.3 Using ACLs For Access Control

What if, in the tradition of old British clubs, Chez Andrea gives each member his own private room, in addition to access to the library, the dining room, the billiard parlor, and other shared spaces? In this case, we need to ensure not just that only members get into the club at all, but that Ken Thompson (known to be a bit of a scamp [T84]) can't slip into Whitfield Diffie's room and short-sheet his bed. We could have one big access control list that specifies allowable access to every room, but that would get unmanageable. Instead, why not have one ACL for each room in the club?

We do the same thing with files in a typical OS that relies on ACLs for access control. Each file has its own access control list, resulting in simpler, shorter lists and quicker access control checks. So our `open()` call in an ACL system will examine a list for `/tmp/foo`, not an ACL encoding all accesses for every file in the system.

When this `open()` call traps to the operating system, the OS consults the running process's PCB to determine who owns the process. That data structure indicates that user X owns the process. The system then must get hold of the access control list for `/tmp/foo`. This ACL is more file metadata, akin to the things we discussed in the chapter titled "Files and Directories." So it's likely to be stored with or near the rest of the metadata for this file. Somehow, we obtain that list from persistent storage. We now look up X on the list. Either X is there or isn't. If not, no access for X. If yes, we'll typically go a step further to determine if the ACL entry for X allows the type of access being requested. In our example, X wanted to open `/tmp/foo` for read and write. Perhaps the ACL allows X to open that file for read, but not for write. In that case, the system will deny the access and return an error to the process.

In principle, this isn't too complicated, but remember the devil being in the details? He's still there. Consider some of those details. For example, where exactly is the ACL persistently stored? It really does need to be persistent for most resources, since the ACLs effectively encode our chosen security policy, which is probably not changing very often. So it's somewhere on the flash drive or disk. Unless it's cached, we'll need to read it off that device every time someone tries to open the file. In most file systems, as was discussed in the sections on persistence, you already need to perform several device reads to actually obtain any information from a file. Are we going to require another read to also get the ACL for the file? If so, where on the device do we put the ACL to ensure that it's quick to access? It would be best if it was close to, or even part of, something we're already reading, which suggests a few possible locations: the file's directory entry, the file's inode, or perhaps the first data block of the file. At the minimum, we want to have the ACL close to one of those locations, and it might be better if it was actually in one of them, such as the inode.

That leads to another vexing detail: how big is this list? If we do the obvious thing and create a list of actual user IDs and access modes, in principle the list could be of arbitrary size, up to the number of users known to the system. For some systems, that could be thousands of entries. But typically files belong to one user and are often available only to that user and perhaps a couple friends. So we wouldn't want to reserve enough space in every ACL for every possible user to be listed, since most users wouldn't appear in most ACLs. With some exceptions, of course: a lot of files should be available in some mode (perhaps read or execute) to all users. After all, commonly used executables (like `ls` and `mv`) are stored in files, and we'll be applying access control to them, just like any other file. Our users will share the same font files, configuration files for networking, and so forth. We have to allow all users to access these files or they won't be able to do much of anything on the system.

So the obvious implementation would reserve a big per-file list that would be totally filled for some files and nearly empty for others. That's clearly wasteful. For the totally filled lists, there's another worrying detail: every time we want to check access in the list, we'll need to search it. Modern computers can search a list of a thousand entries rather quickly, but if we need to perform such searches all the time, we'll add a lot of undesirable overhead to our system. We could solve the problem with variable-sized access control lists, only allocating the space required for each list. Spend a few moments thinking about how you would fit that kind of metadata into the types of file systems we've studied, and the implications for performance.

Fortunately, in most circumstances we can benefit from a bit of legacy handed down to us from the original Bell Labs Unix system. Back in those primeval days when computer science giants roamed the Earth (or at least certain parts of New Jersey), persistent storage was in short supply and pretty expensive. There was simply no way they could afford to store large ACLs for each file. In fact, when they worked it out, they figured they could afford about nine bits for each file's ACL. Nine bits don't go far, but fortunately those early Unix designers had plenty of cleverness to make up for their lack of hardware. They thought about their problem and figured out that there were effectively three modes of access they cared about (read, write, and execute, for most files), and they could handle most security policies with only three entries on each access control list. Of course, if they were going to use one bit per access mode per entry, they would have already used up their nine bits, leaving no bits to specify who the entry pertained to. So they cleverly partitioned the entries on their access control list into three groups. One is the owner of the file, whose identity they had already stored in the inode. One is the members of a particular group or users; this group ID was also stored in the inode. The final one is everybody else, i.e., everybody who wasn't the owner or a member of his group. No need to use any bits to store that, since it was just the complement of the user and group.

This solution not only solved the problem of the amount of storage eaten up by ACLs, but also solved the problem of the cost of accessing and checking them. You already needed to access a file's inode to do almost anything with it, so if the ACL was embedded in the inode, there would be no extra seeks and reads to obtain it. And instead of a search of an arbitrary sized list, a little simple logic on a few bits would provide the answer to the access control question. And that logic is still providing the answer in most systems that use Posix-compliant file systems to this very day. Of course, the approach has limitations, since it cannot express complex access modes and sharing relationships. For that reason, some modern systems (such as Windows) allow extensions that permit the use of more general ACLs, but many rely on the tried-and-true Unix-style nine-bit ACLs[5].

There are some good features of ACLs and some limiting features. Good points first. First, what if you want to figure out who is allowed to access a resource? If you're using ACLs, that's an easy question to answer, since you can simply look at the ACL itself. Second, if you want to change the set of subjects who can access an object, you merely need to change the ACL, since nothing else can give the user access. Third, since the ACL is typically kept either with or near the file itself, if you can get to the file, you can get to all relevant access control information. This is particularly important in distributed systems, but it also has good performance implications for all systems, as long as your design keeps the ACL near the file or its inode.

Now for the less desirable features. First, ACLs require you to solve problems we mentioned earlier: having to store the access control information somewhere near the file and dealing with potentially expensive searches of long lists. We described some practical solutions that work pretty well in most systems, but these solutions limit what ACLs can do. Second, what if you want to figure out the entire set of resources some principal (a process or a user) is permitted to access? You'll need to check every single ACL in the system, since that principal might be on any of them. Third, in a distributed environment, you need to have a common view of identity across all the machines for ACLs to be effective. If a user on `cs.ucla.edu` wants to access a file stored on `cs.wisconsin.edu`, the Wisconsin machine is going to check some identity provided by UCLA against an access control list stored at Wisconsin. Does user `remzi` at UCLA actually refer to the same principal as user `remzi` at Wisconsin? If not, you may allow a remote user to access something he shouldn't. But trying to maintain a consistent name space of users across multiple different computing domains is challenging.

[5]The history is a bit more complicated than this. The CTSS system offered a more limited form of condensed ACL than Unix did [C+63], and the Multics system included the concept of groups in a more general access control list consisting of character string names of users and groups [S74]. Thus, the Unix approach was a cross-breeding of these even earlier systems.

ASIDE: NAME SPACES

We just encountered one of the interesting and difficult problems in distributed systems: what do names mean on different machines? This **name space** problem is relatively easy on a single computer. If the name chosen for a new thing is already in use, don't allow it to be assigned. So when a particular name is issued on that system by any user or process, it means the same thing. `/etc/password` is the same file for you and for all the other users on your computer.

But what about distributed systems composed of multiple computers? If you want the same guarantee about unique names understood by all, you need to make sure someone on a machine at UCLA does not create a name already being used at the University of Wisconsin. How to do that? Different answers have different pluses and minuses. One approach is not to bother and to understand that the namespaces are different – that's what we do with process IDs, for example. Another approach is to require an authority to approve name selection – that's more or less how AFS handles file name creation. Another approach is to hand out portions of the name space to each participant and allow them to assign any name from that portion, but not any other name – that's how the World Wide Web and the IPv4 address space handle the issue. None of these answers are universally right or wrong. Design your name space for your needs, but understand the implications.

55.4 Using Capabilities For Access Control

Access control lists are not your only option for controlling access in computer systems. Almost, but not quite. You can also use capabilities, the option that's more like keys or tickets. Chez Andrea could give keys to its members to allow admission. Different rooms could have different keys, preventing the more mischievous members from leaving little surprises in other members' rooms. Each member would carry around a set of keys that would admit him or her to the particular areas of the club she should have access to. Like ACLs, capabilities have a long history of use in computer systems, with Dennis and van Horn [DV64] being perhaps the earliest example. Wulf et al. [W+74] describe the Hydra Operating System, which used capabilities as a fundamental control mechanism. Levy [L84] gives a book-length summary of the use of capabilities in early hardware and software systems. In capability systems, a running process has some set of capabilities that specify its access permissions. If you're using a pure capability system, there is no ACL anywhere, and this set is the entire encoding of the access permissions for this process. That's not how Linux or Windows work, but other operating systems, such as Hydra, examined this approach to handling access control.

How would we perform that `open()` call in this kind of pure capability system? When the call is made, either your application would provide a capability permitting your process to open the file in question as a parameter, or the operating system would find the capability for you. In either case, the operating system would check that the capability does or does not allow you to perform a read/write open on file `/tmp/foo`. If it does, the OS opens it for you. If not, back comes an error to your process, chiding it for trying to open a file it does not have a capability for. (Remember, we're not talking about Linux here. Linux uses ACLs, not capabilities, to determine if an `open()` call should be allowed.)

There are some obvious questions here. What, precisely, is a capability? Clearly we're not talking about metal keys or paper tickets. Also, how does the OS check the validity of capability? And where do capabilities come from, in the first place? Just like all other information in a computer, capabilities are bunches of bits. They are data. Given that there are probably lots of resources to protect, and capabilities must be specific to a resource, capabilities are likely to be fairly long, and perhaps fairly complex. But, ultimately, they're just bits. Anything composed of a bunch of bits has certain properties we must bear in mind. For example, anyone can create any bunch of bits they want. There are no proprietary or reserved bit patterns that processes cannot create. Also, if a process has one copy of a particular set of bits, it's trivial to create more copies of it. The first characteristic implies that it's possible for anyone at all to create any capability they want. The second characteristic implies that once someone has a working capability, they can make as many copies of it as they want, and can potentially store them anywhere they want, including on an entirely different machine.

That doesn't sound so good from a security perspective. If a process needs a capability with a particular bit pattern to open `/tmp/foo` for read and write, maybe it can just generate that bit pattern and successfully give itself the desired access to the file. That's not what we're looking for in an access control mechanism. We want capabilities to be unforgeable. Even if we can get around that problem, the ability to copy a capability would suggest we can't take access permission away, once granted, since the process might have copies of the capability stashed away elsewhere[6]. Further, perhaps the process can grant access to another process merely by using IPC to transfer a copy of the capability to that other process.

We typically deal with these issues when using capabilities for access control by never letting a process get its metaphoric hands on any capability. The operating system controls and maintains capabilities, storing them somewhere in its protected memory space. Processes can perform various operations on capabilities, but only with the mediation of the operating system. If, for example, process A wishes to give process B read/write access to file `/tmp/foo` using capabilities, A can't merely

[6]This ability is commonly called **revocation**. Revocation is easy with ACLs, since you just go to the ACL and change it. Depending on implementation, it can be easy or hard for capabilities.

send B the appropriate bit pattern. Instead, A must make a system call requesting the operating system to give the appropriate capability to B. That gives the OS a chance to decide whether its security policy permits B to access `/tmp/foo` and deny the capability transfer if it does not.

So if we want to rely on capabilities for access control, the operating system will need to maintain its own protected capability list for each process. That's simple enough, since the OS already has a per-process protected data structure, the PCB. Slap a pointer to the capability list (stored in kernel memory) into the process' PCB and you're all set. Now when the process attempts to open `/tmp/foo` for read/write, the call traps to the OS, the OS consults the capability list for that process to see if there is a relevant capability for the operation on the list and proceeds accordingly.

In a general system, keeping an on-line capability list of literally everything some principal is permitted to access would incur high overheads. If we used capabilities for file-based access control, a user might have thousands of capabilities, one for each file the user was allowed to access in any way. Generally, if one is using capabilities, the system persistently stores the capabilities somewhere safe, and imports them as needed. So a capability list attached to a process is not necessarily very long, but there is an issue of deciding which capabilities of the immense set users have at their discretion to give to each process they run.

There is another option. Capabilities need not be stored in the operating system. Instead, they can be cryptographically protected. If capabilities are relatively long and are created with strong cryptography, they cannot be guessed in a practical way and can be left in the user's hands. Cryptographic capabilities make most sense in a distributed system, so we'll talk about them in the chapter on distributed system security.

There are good and bad points about capabilities, just as there were for access control lists. With capabilities, it's easy to determine which system resources a given principal can access. Just look through the principal's capability list. Revoking access merely requires removing the capability from the list, which is easy enough if the OS has exclusive access to the capability (but much more difficult if it does not). If you have the capability readily available in memory, it can be quite cheap to check it, particularly since the capability can itself contain a pointer to the data or software associated with the resource it protects. Perhaps merely having such a pointer is the system's core implementation of capabilities.

On the other hand, determining the entire set of principals who can access a resource becomes more expensive. Any principal might have a capability for the resource, so you must check all principals' capability lists to tell. Simple methods for making capability lists short and manageable have not been as well developed as the Unix method of providing short ACLs. Also, the system must be able to create, store, and retrieve capabilities in a way that overcomes the forgery problem, which can be challenging.

One neat aspect of capabilities is that they offer a good way to create processes with limited privileges. With access control lists, a process inherits the identity of its parent process, also inheriting all of the privileges of that principal. It's hard to give the process just a subset of the parent's privileges. Either you need to create a new principal with those limited privileges, change a bunch of access control lists, and set the new process's identity to that new principal, or you need some extension to your access control model that doesn't behave quite the way access control lists ordinarily do. With capabilities, it's easy. If the parent has capabilities for X, Y, and Z, but only wants the child process to have the X and Y capabilities, when the child is created, the parent transfers X and Y, not Z.

In practice, user-visible access control mechanisms tend to use access control lists, not capabilities, for a number of reasons. However, under the covers operating systems make extensive use of capabilities. For example, in a typical Linux system, that `open()` call we were discussing uses ACLs for access control. However, assuming the Linux `open()` was successful, as long as the process keeps the file open, the ACL is not examined on subsequent reads and writes. Instead, Linux creates a data structure that amounts to a capability indicating that the process has read and write privileges for that file. This structure is attached to the process's PCB. On each read or write operation, the OS can simply consult this data structure to determine if reading and writing are allowed, without having to find the file's access control list. If the file is closed, this capability-like structure is deleted from the PCB and the process can no longer access the file without performing another `open()` which goes back to the ACL. Similar techniques can be used to control access to hardware devices and IPC channels, especially since UNIX-like systems treat these resources as if they were files. This combined use of ACLs and capabilities allows the system to avoid some of the problems associated with each mechanism. The cost of checking an access control list on every operation is saved because this form of capability is easy to check, being merely the presence or absence of a pointer in an operating system data structure. The cost of managing capabilities for all accessible objects is avoided because the capability is only set up after a successful ACL check. If the object is never accessed by a process, the ACL is never checked and no capability is required. Since any given process typically opens only a tiny fraction of all the files it is permitted to open, the scaling issue doesn't usually arise.

55.5 Mandatory And Discretionary Access Control

Who gets to decide what the access control on a computer resource should be? For most people, the answer seems obvious: whoever owns the resource. In the case of a user's file, the user should determine access control settings. In the case of a system resource, the system administrator, or perhaps the owner of the computer, should determine them. However, for some systems and some security policies, that's not the right answer. In particular, the parties who care most about information security sometimes want tighter controls than that.

The military is the most obvious example. We've all heard of Top Secret information, and probably all understand that even if you are allowed to see Top Secret information, you're not supposed to let other people see it, too. And that's true even if the information in question is in a file that you created yourself, such as a report that contains statistics or quotations from some other Top Secret document. In these cases, the simple answer of the creator controlling access permissions isn't right. Whoever is in overall charge of information security in the organization needs to make those decisions, which implies that principal has the power to set the access controls for information created by and belonging to other users, and that those users can't override his decisions. The more common case is called **discretionary access control**. Whether almost anyone or almost no one is given access to a resource is at the discretion of the owning user. The more restrictive case is called **mandatory access control**. At least some elements of the access control decisions in such systems are mandated by an authority, who can override the desires of the owner of the information. The choice of discretionary or mandatory access control is orthogonal to whether you use ACLs or capabilities, and is often independent of other aspects of the access control mechanism, such as how access information is stored and handled. A mandatory access control system can also include discretionary elements, which allow further restriction (but not loosening) of mandatory controls.

Many people will never work with a system running mandatory access controls, so we won't go further into how they work, beyond observing that clearly the operating system is going to be involved in enforcing them. Should you ever need to work in an environment where mandatory access control is important, you can be sure you will hear about it. You should learn more about it at that point, since when someone cares enough to use mandatory access control mechanisms, they also care enough to punish users who don't follow the rules. Loscocco [L01] describes a special version of Linux that incorporates mandatory access control. This is a good paper to start with if you want to learn more about the characteristics of such systems.

55.6 Practicalities Of Access Control Mechanisms

Most systems expose either a simple or more powerful access control list mechanism to their users, and most of them use discretionary access control. However, given that a modern computer can easily have hundreds of thousands, or even millions of files, having human users individually set access control permissions on them is infeasible. Generally, the system allows each user to establish a default access permission that is used for every file he creates. If one uses the Linux `open()` call to create a file, one can specify which access permissions to initially assign to that file. Access permissions on newly created files in Unix/Linux systems can be further controlled by the **`umask()`** call, which applies to all new file creations by the process that performed it.

ASIDE: THE ANDROID ACCESS CONTROL MODEL

The Android system is one of the leading software platforms for today's mobile computing devices, especially smart phones. These devices pose different access control challenges than classic server computers, or even personal desktop computers or laptops. Their functionality is based on the use of many relatively small independent applications, commonly called apps, that are downloaded, installed, and run on a device belonging to only a single user. Thus, there is no issue of protecting multiple users on one machine from each other. If one used a standard access control model, these apps would run under that user's identity. But apps are developed by many entities, and some may be malicious. Further, most apps have no legitimate need for most of the resources on the device. If they are granted too many privileges, a malicious app can access the phone owner's contacts, make phone calls, or buy things over the network, among many other undesirable behaviors. The principle of least privilege implies that we should not give apps the full privileges belonging to owner, but they must have some privileges if they are to do anything interesting.

Android runs on top of a version of Linux, and an application's access limitations are achieved in part by generating a new user ID for each installed app. The app runs under that ID and its accesses can be controlled on that basis. However, the Android middleware offers additional facilities for controlling access. Application developers define accesses required by their app. When a user considers installing an app on their device, they are shown what permissions it requires. The user can either grant the app those permissions, not install the app, or limit its permissions, though the latter choice may also limit app utility. Also, the developer specifies ways in which other apps can communicate with the new app. The data structure used to encode this access information is called a **permission label**. An app's permission labels (both what it can access and what it provides to others) are set at app design time, and encoded into a particular Android system at the moment the app is installed on that machine.

Permission labels are thus like capabilities, since possession of them by the app allows the app to do something, while lacking a label prevents the app from doing that thing. An app's set of permission labels is set statically at install time. The user can subsequently change those permissions, although limiting them may damage app functionality. Permission labels are a form of mandatory access control. The Android security model is discussed in detail by Enck et al. [E+09].

The Android security approach is interesting, but not perfect. In particular, users are not always aware of the implications of granting an application access to something, and, faced with the choice of granting the access or not being able to effectively use the app, they will often grant it. This behavior can be problematic, if the app is malicious.

If desired, the owner can alter that initial ACL, but experience shows that users rarely do. This tendency demonstrates the importance of properly chosen defaults. Here, as in many other places in an operating system, a theoretically changeable or tunable setting will, in practice, be used unaltered by almost everyone almost always.

However, while many will never touch access controls on their resources, for an important set of users and systems these controls are of vital importance to achieve their security goals. Even if you mostly rely on defaults, many software installation packages use some degree of care in setting access controls on executables and configuration files they create. Generally, you should exercise caution in fiddling around with access controls in your system. If you don't know what you're doing, you might expose sensitive information or allow attackers to alter critical system settings. If you tighten existing access controls, you might suddenly cause a bunch of daemon programs running in the background to stop working.

One practical issue that many large institutions discovered when trying to use standard access control methods to implement their security policies is that people performing different roles within the organization require different privileges. For example, in a hospital, all doctors might have a set of privileges not given to all pharmacists, who themselves have privileges not given to the doctors. Organizing access control on the basis of such roles and then assigning particular users to the roles they are allowed to perform makes implementation of many security policies easier. This approach is particularly valuable if certain users are permitted to switch roles depending on the task they are currently performing, since then one need not worry about setting or changing the individual's access permissions on the fly, but simply switch their role from one to another. Usually they will hold the role's permission only as long as they maintain that role. Once they exit the particular role (perhaps to enter a different role with different privileges), they lose the privileges of the role they exit.

This observation led to the development of **Role-Based Access Control**, or **RBAC**. The core ideas had been around for some time before they were more formally laid out in a research paper by Ferraiolo and Kuhn [FK92]. Now RBAC is in common use in many organizations, particularly large ones. Large organizations face more serious management challenges than small ones, so approaches like RBAC that allow groups of users to be dealt with in one operation can significantly ease the management task. For example, if a company determines that all programmers should be granted access to a new library that has been developed, but accountants should not, RBAC would achieve this effect with a single operation that assigns the necessary privilege to the *Programmer* role. If a programmer is promoted to a management position for which access to the library is unnecessary, the company can merely remove the *Programmer* role from the set of roles the manager could take on.

Such restrictions do not necessarily imply that you suspect your accountants of being dishonest and prone to selling your secret library code to competitors[7]. Remember the principle of least privilege: when you give someone access to something, you are relying not just on their honesty, but on their caution. If accountants can't access the library at all,

[7]Dishonest accountants are generally good to avoid, so you probably did your best to hire honest ones, after all. Unless you're Bernie Madoff [W20], perhaps...

then neither malice nor carelessness on their part can lead to an accountant's privileges leaking your library code. Least privilege is not just a theoretically good idea, but a vital part of building secure systems in the real world.

RBAC sounds a bit like using groups in access control lists, and there is some similarity, but RBAC systems are a good deal more powerful than mere group access permissions; RBAC systems allow a particular user to take on multiple disjoint roles. Perhaps our programmer was promoted to a management position, but still needs access to the library, for example when another team member's code needs to be tested. An RBAC system would allow our programmer to switch between the role of manager and programmer, temporarily leaving behind rights associated with the manager and gaining rights associated with the programmer role. When the manager tested someone else's new code, the manager would have permission to access the library, but would *not* have permission to access team member performance reviews. Thus, if a sneaky programmer slipped malicious code into the library (e.g., that tried to read other team members' performance reviews, or learn their salaries), the manager running that code would not unintentionally leak that information; using the proper role at the proper time prevents it.

These systems often require a new authentication step to take on an RBAC role, and usually taking on Role A requires relinquishing privileges associated with one's previous role, say Role B. The manager's switch to the code testing role would result in temporarily relinquishing privileges to examine the performance reviews. On completing the testing, the manager would switch back to the role allowing access to the reviews, losing privilege to access the library. RBAC systems may also offer finer granularity than merely being able to read or write a file. A particular role (*Salesperson*, for instance) might be permitted to add a purchase record for a particular product to a file, but would not be permitted to add a re-stocking record for the same product to the same file, since salespeople don't do re-stocking. This degree of control is sometimes called **type enforcement**. It associates detailed access rules to particular objects using what is commonly called a **security context** for that object. How exactly this is done has implications for performance, storage of the security context information, and authentication.

One can build a very minimal RBAC system under Linux and similar OSes using ACLs and groups. These systems have a feature in their access control mechanism called **privilege escalation**. Privilege escalation allows careful extension of privileges, typically by allowing a particular program to run with a set of privileges beyond those of the user who invokes them. In Unix and Linux systems, this feature is called **`setuid`**, and it allows a program to run with privileges associated with a different user, generally a user who has privileges not normally available to the user who runs the program. However, those privileges are only granted during the run of that program and are lost when the program exits. A carefully written `setuid` program will only perform a limited set of oper-

TIP: PRIVILEGE ESCALATION CONSIDERED DANGEROUS

We just finished talking about how we could use privilege escalation to temporarily change what one of our users can do, and how this offers us new security options. But there's a dangerous side to privilege escalation. An attacker who breaks into your system frequently compromises a program running under an identity with very limited privileges. Perhaps all it's supposed to be able to do is work with a few simple informational files and provide remote users with their content, and maybe run standard utilities on those files. It might not even have write access to its files. You might think that this type of compromise has done little harm to the system, since the attacker cannot use the access to do very much.

This is where the danger of privilege escalation comes into play. Attackers who have gained any kind of a foothold on a system will then look around for ways to escalate their privileges. Even a fairly unprivileged application can do a lot of things that an outsider cannot directly do, so attackers look for flaws in the code or configuration that the compromised application can access. Such attempts to escalate privilege are usually an attacker's first order of business upon successful compromise of a system.

In many systems, there is a special user, often called the **superuser** or **root** user. This user has a lot more privilege than any other user on the system, since its purpose is to allow for the most vital and far-reaching system administration changes on that system. The paramount goal of an attacker with a foothold on your system is to use privilege escalation to become the root user. An attacker who can do that will effectively have total control of your system. Such an attacker can look at any file, alter any program, change any configuration, and perhaps even install a different operating system. This danger should point out how critical it is to be careful in allowing any path that permits privilege escalation up to superuser privilege.

ations using those privileges, ensuring that privileges cannot be abused[8]. One could create a simple RBAC system by defining an artificial user for each role and associating desired privileges with that user. Programs using those privileges could be designated as `setuid` to that user.

The Linux `sudo` command, which we encountered in the authentication chapter, offers this kind of functionality, allowing some designated users to run certain programs under another identity. For example,

```
sudo -u Programmer install newprogram
```

would run this install command under the identity of user `Programmer`, rather than the identity of the user who ran the command, assuming that user was on a system-maintained list of users allowed to take on the identity `Programmer`. Secure use of this approach requires careful configura-

[8] Unfortunately, not all programs run with the `setuid` feature are carefully written, which has led to many security problems over the years. Perhaps true for all security features, alas?

tion of system files controlling who is allowed to execute which programs under which identities. Usually the `sudo` command requires a new authentication step, as with other RBAC systems.

For more advanced purposes, RBAC systems typically support finer granularity and more careful tracking of role assignment than `setuid` and `sudo` operations allow. Such an RBAC system might be part of the operating system or might be some form of add-on to the system, or perhaps a programming environment. Often, if you're using RBAC, you also run some degree of mandatory access control. If not, in the example of `sudo` above, the user running under the `Programmer` identity could run a command to change the access permissions on files, making the `install` command available to non- programmers. With mandatory access control, a user could take on the role of `Programmer` to do the installation, but could not use that role to allow salespeople or accountants to perform the installation.

55.7 Summary

Implementing most security policies requires controlling which users can access which resources in which ways. Access control mechanisms built in to the operating system provide the necessary functionality. A good access control mechanism will provide complete mediation (or close to it) of security-relevant accesses through use of a carefully designed and implemented reference monitor.

Access control lists and capabilities are the two fundamental mechanisms used by most access control systems. Access control lists specify precisely which subjects can access which objects in which ways. Presence or absence on the relevant list determines if access is granted. Capabilities work more like keys in a lock. Possession of the correct capability is sufficient proof that access to a resource should be permitted. User-visible access control is more commonly achieved with a form of access control list, but capabilities are often built in to the operating system at a level below what the user sees. Neither of these access control mechanisms is inherently better or worse than the other. Rather, like so many options in system design, they have properties that are well suited to some situations and uses and poorly suited to others. You need to understand how to choose which one to use in which circumstance.

Access control mechanisms can be discretionary or mandatory. Some systems include both. Enhancements like type enforcement and role-based access control can make it easier to achieve the security policy you require.

Even if the access control mechanism is completely correct and extremely efficient, it can do no more than implement the security policies that it is given. Security failures due to faulty access control mechanisms are rare. Security failures due to poorly designed policies implemented by those mechanisms are not.

References

[C+63] "The Compatible Time Sharing System: A Programmer's Guide" by F. J. Corbato, M. M. Daggett, R. C. Daley, R. J. Creasy, J. D. Hellwig, R. H. Orenstein, and L. K. Korn. M.I.T. Press, 1963. *The programmer's guide for the early and influential CTSS time sharing system. Referenced here because it used an early version of an access control list approach to protecting data stored on disk.*

[DV64] "Programming Semantics for Multiprogrammed Computations" by Jack B. Dennis and Earl. C. van Horn. Communications of the ACM, Vol. 9, No. 3, March 1966. *The earliest discussion of the use of capabilities to perform access control in a computer. Though the authors themselves point to the "program reference table" used in the Burroughs B5000 system as an inspiration for this notion.*

[E+09] "Understanding Android Security" by William Enck, Machigar Ongtang, and Patrick McDaniel. IEEE Security and Privacy, Vol. 7, No. 1, January/February 1999. *An interesting approach to providing access control in a particular and important kind of machine. The approach has not been uniformly successful, but it is worth understanding in more detail than we discuss in this chapter.*

[FK92] "Role-Based Access Controls" by David Ferraiolo and D. Richard Kuhn. 15th National Computer Security Conference, October 1992. *The concepts behind RBAC were floating around since at least the 70s, but this paper is commonly regarded as the first discussion of RBAC as a formal concept with particular properties.*

[L84] "Capability-Based Computer Systems" by Henry Levy. Digital Press, 1984. *A full book on the use of capabilities in computer systems, as of 1984. It includes coverage of both hardware using capabilities and operating systems, like Hydra, that used them.*

[L01] "Integrating Flexible Support for Security Policies Into the Linux Operating System" by Peter Loscocco. Proceedings of the FREENIX Track at the USENIX Annual Technical Conference 2001. *The NSA built this version of Linux that incorporates mandatory access control and other security features into Linux. A good place to dive into the world of mandatory access control, if either necessity or interest motivates you to do so.*

[S74] "Protection and Control of Information Sharing in Multics" by Jerome Saltzer. Communications of the ACM, Vol. 17, No. 7, July 1974. *Sometimes it seems that every system idea not introduced in CTSS was added in Multics. In this case, it's the general use of groups in access control lists.*

[T84] "Reflections on Trusting Trust" by Ken Thompson. Communications of the ACM, Vol. 27, No. 8, August 1984. *Ken Thompson's Turing Award lecture, in which he pointed out how sly systems developers can slip in backdoors without anyone being aware of it. People have wondered ever since if he actually did what he talked about...*

[W20] "Bernie Madoff" by Wikipedia. `https://en.wikipedia.org/wiki/Bernie_Madoff`. *Bernie Madoff (painfully, pronounced "made off", as in "made off with your money") built a sophisticated Ponzi scheme, a fraud of unimaginable proportions (nearly 100 billion dollars). He is, as Wikipedia says, an "American charlatan". As relevant here, he probably hired dishonest accountants, or was one himself.*

[W+74] "Hydra: The Kernel of a Multiprocessor Operating System" by W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pearson, and F. Pollack. Communications of the ACM, Vol. 17, No. 6, June 1974. *A paper on a well-known operating system that made extensive and sophisticated use of capabilities to handle access control.*

56

Protecting Information With Cryptography

Chapter by **Peter Reiher (UCLA)**

56.1 Introduction

In previous chapters, we've discussed clarifying your security goals, determining your security policies, using authentication mechanisms to identify principals, and using access control mechanisms to enforce policies concerning which principals can access which computer resources in which ways. While we identified a number of shortcomings and problems inherent in all of these elements of securing your system, if we regard those topics as covered, what's left for the operating system to worry about, from a security perspective? Why isn't that everything?

There are a number of reasons why we need more. Of particular importance: not everything is controlled by the operating system. But perhaps you respond, you told me the operating system is all-powerful! Not really. It has substantial control over a limited domain – the hardware on which it runs, using the interfaces of which it is given control. It has no real control over what happens on other machines, nor what happens if one of its pieces of hardware is accessed via some mechanism outside the operating system's control.

But how can we expect the operating system to protect something when the system does not itself control access to that resource? The answer is to prepare the resource for trouble in advance. In essence, we assume that we are going to lose the data, or that an opponent will try to alter it improperly. And we take steps to ensure that such actions don't cause us problems. The key observation is that if an opponent cannot understand the data in the form it is obtained, our secrets are safe. Further, if the attacker cannot understand it, it probably can't be altered, at least not in a controllable way. If the attacker doesn't know what the data means, how can it be changed into something the attacker prefers?

The core technology we'll use is **cryptography**, a set of techniques to convert data from one form to another, in controlled ways with expected outcomes. We will convert the data from its ordinary form into another form using cryptography. If we do it right, the opponent will not be able to determine what the original data was by examining the protected form.

Of course, if we ever want to use it again ourselves, we must be able to reverse that transformation and return the data to its ordinary form. That must be hard for the opponent to do, as well. If we can get to that point, we can also provide some protection for the data from alteration, or, more precisely, prevent opponents from altering the data to suit their desires, and even know when opponents have tampered with our data. All through the joys of cryptography!

But using cryptography properly is not easy, and many uses of cryptography are computationally expensive. So we need to be selective about where and when we use cryptography, and careful in how we implement it and integrate it into our systems. Well chosen uses that are properly performed will tremendously increase security. Poorly chosen uses that are badly implemented won't help at all, and may even hurt.

THE CRUX OF THE PROBLEM:
HOW TO PROTECT INFORMATION OUTSIDE THE OS'S DOMAIN

How can we use cryptography to ensure that, even if others gain access to critical data outside the control of the operating system, they will be unable to either use or alter it? What cryptographic technologies are available to assist in this problem? How do we properly use those technologies? What are the limitations on what we can do with them?

56.2 Cryptography

Many books have been written about cryptography, but we're only going to spend a chapter on it. We'll still be able to say useful things about it because, fortunately, there are important and complex issues of cryptography that we can mostly ignore. That's because we aren't going to become cryptographers ourselves. We're merely going to be users of the technology, relying on experts in that esoteric field to provide us with tools that we can use without having full understanding of their workings[1]. That sounds kind of questionable, but you are already doing just that. Relatively few of us really understand the deep details of how our computer hardware works, yet we are able to make successful use of it, because we have good interfaces and know that smart people have taken great care in building the hardware for us. Similarly, cryptography provides us with strong interfaces, well-defined behaviors, and better than usual assurance that there is a lot of brain power behind the tools we use.

That said, cryptography is no magic wand, and there is a lot you need to understand merely to use it correctly. That, particularly in the context of operating system use, is what we're going to concentrate on here.

[1] If you'd like to learn more about the fascinating history of cryptography, check out Kahn [K96]. If more technical detail is your desire, Schneier [S96] is a good start.

The basic idea behind cryptography is to take a piece of data and use an algorithm (often called a **cipher**), usually augmented with a second piece of information (which is called a **key**), to convert the data into a different form. The new form should look nothing like the old one, but, typically, we want to be able to run another algorithm, again augmented with a second piece of information, to convert the data back to its original form.

Let's formalize that just a little bit. We start with data P (which we usually call the **plaintext**), a key K, and an encryption algorithm $E()$. We end up with C, the altered form of P, which we usually call the **ciphertext**:

$$C = E(P, K) \tag{56.1}$$

For example, we might take the plaintext "Transfer $100 to my savings account" and convert it into ciphertext "Sqzmredq #099 sn lx rzuhmfr zbbntms." This example actually uses a pretty poor encryption algorithm called a Caesar cipher. Spend a minute or two studying the plaintext and ciphertext and see if you can figure out what the encryption algorithm was in this case.

The reverse transformation takes C, which we just produced, a decryption algorithm $D()$, and the key K:

$$P = D(C, K) \tag{56.2}$$

So we can decrypt "Sqzmredq #099 sn lx rzuhmfr zbbntms" back into "Transfer $100 to my savings account." If you figured out how we encrypted the data in the first place, it should be easy to figure out how to decrypt it.

We use cryptography for a lot of things, but when discussing it generally, it's common to talk about messages being sent and received. In such discussions, the plaintext P is the message we want to send and the ciphertext C is the protected version of that message that we send out into the cold, cruel world.

For the encryption process to be useful, it must be deterministic, so the first transformation always converts a particular P using a particular K to a particular C, and the second transformation always converts a particular C using a particular K to the original P. In many cases, $E()$ and $D()$ are actually the same algorithm, but that is not required. Also, it should be very hard to figure out P from C without knowing K. Impossible would be nice, but we'll usually settle for computationally infeasible. If we have that property, we can show C to the most hostile, smartest opponent in the world and they still won't be able to learn what P is.

Provided, of course, that ...

This is where cleanly theoretical papers and messy reality start to collide. We only get that pleasant assurance of secrecy if the opponent does not know both $D()$ and our key K. If they are known, the opponent will apply $D()$ and K to C and extract the same information P that we can.

It turns out that we usually can't keep $E()$ and $D()$ secret. Since we're not trying to be cryptographers, we won't get into the why of the matter, but it is extremely hard to design good ciphers. If the cipher has weaknesses, then an opponent can extract the plaintext P even without K. So we need to have a really good cipher, which is hard to come by. Most of us don't have a world-class cryptographer at our fingertips to design a new one, so we have to rely on one of a relatively small number of known strong ciphers. AES, a standard cipher that was carefully designed and thoroughly studied, is one good example that you should think about using.

It sounds like we've thrown away half our protection, since now the cryptography's benefit relies entirely on the secrecy of the key. Precisely. Let's say that again in all caps, since it's so important that you really need to remember it: **THE CRYPTOGRAPHY'S BENEFIT RELIES ENTIRELY ON THE SECRECY OF THE KEY**. It probably wouldn't hurt for you to re-read that statement a few dozen times, since the landscape is littered with insecure systems that did not take that lesson to heart.

The good news is that if you're using a strong cipher and are careful about maintaining key secrecy, your cryptography is strong. You don't need to worry about anything else. The bad news is that maintaining key secrecy in practical systems for real uses of cryptography isn't easy. We'll talk more about that later.

For the moment, revel in the protection we have achieved, and rejoice to learn that we've gotten more than secrecy from our proper use of cryptography! Consider the properties of the transformations we've performed. If our opponent gets access to our encrypted data, it can't be understood. But what if the opponent can alter it? What's being altered is the encrypted form, i.e., making some changes in C to convert it to, say, C'. What will happen when we try to decrypt C? Well, it won't decrypt to P. It will decrypt to something else, say P'. For a good cipher of the type you should be using, it will be difficult to determine what a piece of ciphertext C' will decrypt to, unless you know K. That means it will be hard to predict which ciphertext you need to have to decrypt to a particular plaintext. Which in turn means that the attacker will have no idea what the altered ciphertext C' will decrypt to.

Out of all possible bit patterns it could decrypt to, the chances are good that P' will turn out to be garbage, when considered in the context of what we expected to see: ASCII text, a proper PDF file, or whatever. If we're careful, we can detect that P' isn't what we started with, which would tell us that our opponent tampered with our encrypted data. If we want to be really sure, we can perform a hashing function on the plaintext and include the hash with the message or encrypted file. If the plaintext we get out doesn't produce the same hash, we will have a strong indication that something is amiss.

So we can use cryptography to help us protect the integrity of our data, as well.

TIP: DEVELOPING YOUR OWN CIPHERS: DON'T DO IT

Don't.

It's tempting to leave it at that, since it's really important that you follow this guidance. But you may not believe it, so we'll expand a little. The world's best cryptographers often produce flawed ciphers. Are you one of the world's best cryptographers? If you aren't, and the top experts often fail to build strong ciphers, what makes you think you'll do better, or even as well?

We know what you'll say next: but the cipher I wrote is so strong that I can't even break it myself. Well, pretty much anyone who puts their mind to it can create a cipher they can't break themselves. But remember those world-class cryptographers we talked about? How did they get to be world class? By careful study of the underpinnings of cryptography and by breaking other people's ciphers. They're very good at it, and if it's worth their trouble, they will break yours. They might ignore it if you just go around bragging about your wonderful cipher (since they hear that all the time), but if you actually use it for something important, you will unfortunately draw their attention. Following which your secrets will be revealed, following which you will look foolish for designing your own cipher instead of using something standard like AES, which is easier to do, anyway.

So, don't.

Wait, there's more! What if someone hands you a piece of data that has been encrypted with a key K that is known only to you and your buddy Remzi? You know you didn't create it, so if it decrypts properly using key K, you know that Remzi must have created it. After all, he's the only other person who knew key K, so only he could have performed the encryption. Voila, we have used cryptography for authentication! Unfortunately, cryptography will not clean your room, do your homework for you, or make thousands of julienne fries in seconds, but it's a mighty fine tool, anyway.

The form of cryptography we just described is often called **symmetric cryptography**, because the same key is used to encrypt and decrypt the data. For a long time, everyone believed that was the only form of cryptography possible. It turns out everyone was wrong.

56.3 Public Key Cryptography

When we discussed using cryptography for authentication, you might have noticed a little problem. In order to verify the authenticity of a piece of encrypted information, you need to know the key used to encrypt it. If we only care about using cryptography for authentication, that's inconvenient. It means that we need to communicate the key we're using for that purpose to whoever might need to authenticate us. What if we're Microsoft, and we want to authenticate ourselves to every user who has purchased our software? We can't use just one key to do this, because we'd need to send that key to hundreds of millions of users and, once they had that key, they could pretend to be Microsoft by using it to encrypt information. Alternately, Microsoft could generate a different key for each of those hundreds of millions of users, but that would require secretly delivering a unique key to hundreds of millions of users, not to mention keeping track of all those keys. Bummer.

Fortunately, our good friends, the cryptographic wizards, came up with a solution. What if we use two different keys for cryptography, one to encrypt and one to decrypt? Our encryption operation becomes

$$C = E(P, K_{encrypt}) \tag{56.3}$$

And our decryption operation becomes

$$P = D(C, K_{decrypt}) \tag{56.4}$$

Life has just become a lot easier for Microsoft. They can tell everyone their decryption key $K_{decrypt}$, but keep their encryption key $K_{encrypt}$ secret. They can now authenticate their data by encrypting it with their secret key, while their hundreds of millions of users can check the authenticity using the key Microsoft made public. For example, Microsoft could encrypt an update to their operating system with $K_{encrypt}$ and send it out to all their users. Each user could decrypt it with $K_{decrypt}$. If it decrypted into a properly formatted software update, the user could be sure it was created by Microsoft. Since no one else knows that private key, no one else could have created the update.

Sounds like magic, but it isn't. It's actually mathematics coming to our rescue, as it so frequently does. We won't get into the details here, but you have to admit it's pretty neat. This form of cryptography is called **public key cryptography**, since one of the two keys can be widely known to the entire public, while still achieving desirable results. The key everyone knows is called the **public key**, and the key that only the owner knows is called the **private key**. Public key cryptography (often abbreviated as **PK**) has a complicated invention history, which, while interesting, is not really germane to our discussion. Check out a paper by a pioneer in the field, Whitfield Diffie, for details [D88].

Public key cryptography avoids one hard issue that faced earlier forms of cryptography: securely distributing a secret key. Here, the private key is created by one party and kept secret by him. It's never distributed to anyone else. The public key must be distributed, but generally we don't care if some third party learns this key, since they can't use it to sign messages. Distributing a public key is an easier problem than distributing a secret key, though, alas, it's harder than it sounds. We'll get to that.

Public key cryptography is actually even neater, since it works the other way around. You can use the decryption key $K_{decrypt}$ to encrypt, in which case you need the encryption key $K_{encrypt}$ to decrypt. We still expect the encryption key to be kept secret and the decryption key to be publicly known, so doing things in this order no longer allows authentication. Anyone could encrypt with $K_{decrypt}$, after all. But only the owner of the key can decrypt such messages using $K_{encrypt}$. So that allows anyone to send an encrypted message to someone who has a private key, provided you know their public key. Thus, PK allows authentication if you encrypt with the private key and secret communication if you encrypt with the public key.

What if you want both, as you very well might? You'll need two different key pairs to do that. Let's say Alice wants to use PK to communicate secretly with her pal Bob, and also wants to be sure Bob can authenticate her messages. Let's also say Alice and Bob each have their own PK pair. Each of them knows his or her own private key and the other party's public key. If Alice encrypts her message with her own private key, she'll authenticate the message, since Bob can use her public key to decrypt and will know that only Alice could have created that message. But everyone knows Alice's public key, so there would be no secrecy achieved. However, if Alice takes the authenticated message and encrypts it a second time, this time with Bob's public key, she will achieve secrecy as well. Only Bob knows the matching private key, so only Bob can read the message. Of course, Bob will need to decrypt twice, once with his private key and then a second time with Alice's public key.

Sounds expensive. It's actually worse than you think, since it turns out that public key cryptography has a shortcoming: it's much more computationally expensive than traditional cryptography that relies on a single shared key. Public key cryptography can take hundreds of times longer to perform than standard symmetric cryptography. As a result, we really can't afford to use public key cryptography for everything. We need to pick and choose our spots, using it to achieve the things it's good at.

There's another important issue. We rather blithely said that Alice knows Bob's public key and Bob knows Alice's. How did we achieve this blissful state of affairs? Originally, only Alice knew her public key and only Bob knew his public key. We're going to need to do something to get that knowledge out to the rest of the world if we want to benefit from the magic of public key cryptography. And we'd better be careful about it, since Bob is going to assume that messages encrypted with the public key he thinks belongs to Alice *were* actually created by Alice. What if some evil genius, called, perhaps, Eve, manages to convince Bob that Eve's public key actually belongs to Alice? If that happens, messages created by Eve would be misidentified by Bob as originating from Alice, subverting our entire goal of authenticating the messages. We'd better make sure Eve can't fool Bob about which public key belongs to Alice.

This leads down a long and shadowy road to the arcane realm of key distribution infrastructures. You will be happier if you don't try to travel that road yourself, since even the most well prepared pioneers who have hazarded it often come to grief. We'll discuss how, in practice, we distribute public keys in a chapter on distributed system security. For the moment, bear in mind that the beautiful magic of public key cryptography rests on the grubby and uncertain foundation of key distribution.

One more thing about PK cryptography: **THE CRYPTOGRAPHY'S BENEFIT RELIES ENTIRELY ON THE SECRECY OF THE KEY**. (Bet you've heard that before.) In this case, the private key. But the secrecy of that private key is every bit as important to the overall benefit of public key cryptography as the secrecy of the single shared key in the case of symmetric cryptography. Never divulge private keys. Never share private keys. Take great care in your use of private keys and in how you store them. If you lose a private key, everything you used it for is at risk, and whoever gets hold of it can pose as you and read your secret messages. That wouldn't be very good, would it?

56.4 Cryptographic Hashes

As we discussed earlier, we can protect data integrity by using cryptography, since alterations to encrypted data will not decrypt properly. We can reduce the costs of that integrity check by hashing the data and encrypting just the hash, instead of encrypting the entire thing. However, if we want to be really careful, we can't use just any hash function, since hash functions, by their very nature, have **hash collisions**, where two different bit patterns hash to the same thing. If an attacker can change the bit pattern we intended to send to some other bit pattern that hashes to the same thing, we would lose our integrity property.

So to be particularly careful, we can use a **cryptographic hash** to ensure integrity. Cryptographic hashes are a special category of hash functions with several important properties:

- It is computationally infeasible to find two inputs that will produce the same hash value.
- Any change to an input will result in an unpredictable change to the resulting hash value.
- It is computationally infeasible to infer any properties of the input based only on the hash value.

Based on these properties, if we only care about data integrity, rather than secrecy, we can take the cryptographic hash of a piece of data, encrypt only that hash, and send both the encrypted hash and the unencrypted data to our partner. If an opponent fiddles with the data in transit, when we decrypt the hash and repeat the hashing operation on the data, we'll see a mismatch and detect the tampering[2].

[2]Why do we need to encrypt the cryptographic hash? Well, anyone, including our opponent, can run a cryptographic hashing algorithm on anything, including an altered version of the message. If we don't encrypt the hash, the attacker will change the message, compute a new hash, replace both the original message and the original hash with these versions, and send the result. If the hash we sent is encrypted, though, the attacker can't know what the encrypted version of the altered hash should be.

To formalize it a bit, to perform a cryptographic hash we take a plaintext P and a hashing algorithm $H()$. Note that there is not necessarily any key involved. Here's what happens:

$$S = H(P) \tag{56.5}$$

Since cryptographic hashes are a subclass of hashes in general, we normally expect S to be shorter than P, perhaps a lot shorter. That implies there will be collisions, situations in which two different plaintexts P and P' both hash to S. However, the properties of cryptographic hashes outlined above will make it difficult for an adversary to make use of collisions. Even if you know both S and P, it should be hard to find any other plaintext P' that hashes to S[3]. It won't be hard to figure out what S′ should be for an altered value of plaintext P', since you can simply apply the cryptographic hashing algorithm directly to P'. But even a slightly altered version of P, such as a P' differing only in one bit, should produce a hash S′ that differs from S in completely unpredictable ways.

Cryptographic hashes can be used for other purposes than ensuring integrity of encrypted data, as well. They are the class of hashes of choice for storing salted hashed passwords, for example, as discussed in the chapter on authentication. They can be used to determine if a stored file has been altered, a function provided by well-known security software like Tripwire. They can also be used to force a process to perform a certain amount of work before submitting a request, an approach called "proof of work." The submitter is required to submit a request that hashes to a certain value using some specified cryptographic hash, which, because of the properties of such hashes, requires them to try a lot of request formats before finding one that hashes to the required value. Since each hash operation takes some time, submitting a proper request will require a predictable amount of work. This use of hashes, in varying forms, occurs in several applications, including spam prevention and blockchains.

Like other cryptographic algorithms, you're well advised to use standard algorithms for cryptographic hashing. For example, the SHA-3 algorithm is commonly regarded as a good choice. However, there is a history of cryptographic hashing algorithms becoming obsolete, so if you are designing a system that uses one, it's wise to first check to see what current recommendations are for choices of such an algorithm.

56.5 Cracking Cryptography

Chances are that you've heard about people cracking cryptography. It's a popular theme in film and television. How worried should you be about that?

[3]Every so often, a well known cryptographic hashing function is "broken" in the sense that someone figures out how to create a P' that uses the function to produce the same hash as P. That happened to a hashing function known as SHA-1 in 2017, rendering that function unsafe and unusable for integrity purposes [G17].

Well, if you didn't take our earlier advice and went ahead and built your own cipher, you should be very worried. Worried enough that you should stop reading this, rip out your own cipher from your system, and replace it with a well-known respected standard. Go ahead, we'll still be here when you get back.

What if you did use one of those standards? In that case, you're probably OK. If you use a modern standard, with a few unimportant exceptions, there are no known ways to read data encrypted with these algorithms without obtaining the key. Which isn't to say your system is secure, but probably no one will break into it by cracking the cryptographic algorithm.

How will they do it, then? Probably by exploiting software flaws in your system having nothing to do with the cryptography, but there's some chance they will crack it by obtaining your keys or exploiting some other flaw in your management of cryptography. How? Software flaws in how you create and use your keys are a common problem. In distributed environments, flaws in the methods used to share keys are also a common weakness that can be exploited. Peter Gutmann produced a nice survey of the sorts of problems improper management of cryptography frequently causes [G02]. Examples include distributing secret keys in software shared by many people, incorrectly transmitting plaintext versions of keys across a network, and choosing keys from a seriously reduced set of possible choices, rather than the larger theoretically possible set. More recently, the Heartbleed attack demonstrated a way to obtain keys being used in OpenSSL sessions from the memory of a remote computer, which allowed an attacker to decrypt the entire session, despite no flaws in either the cipher itself or its implementation, nor in its key selection procedures. This flaw allowed attackers to read the traffic of something between 1/4 and 1/2 of all sites using HTTPS, the cryptographically protected version of HTTP [D+14].

One way attackers deal with cryptography is by guessing the key. Doing so doesn't actually crack the cryptography at all. Cryptographic algorithms are designed to prevent people who don't know the key from obtaining the secrets. If you know the key, it's not supposed to make decryption hard.

So an attacker could try simply guessing each possible key and trying it. That's called a brute force attack, and it's why you should use long keys. For example, AES keys are at least 128 bits. Assuming you generate your AES key at random, an attacker will need to make 2^{127} guesses at your key, on average, before he gets it right. That's a lot of guesses and will take a lot of time. Of course, if a software flaw causes your system to select one out of thirty two possible AES keys, instead of one out of 2^{128}, a brute force attack may become trivial. Key selection is a big deal for cryptography.

For example, the original 802.11 wireless networking standard included no cryptographic protection of data being streamed through the air. The

TIP: SELECTING KEYS

One important aspect of key secrecy is selecting a good one to begin with. For public key cryptography, you need to run an algorithm to select one of the few possible pairs of keys you will use. But for symmetric cryptography, you are free to select any of the possible keys. How should you choose?

Randomly. If you use any deterministic method to select your key, your opponent's problem of finding out your key has just been converted into a problem of figuring out your method. Worse, since you'll probably generate many keys over the course of time, once he knows your method, he'll get all of them. If you use random chance to generate keys, though, figuring out one of them won't help your opponent figure out any of your other keys. This highly desirable property in a cryptographic system is called **perfect forward secrecy**.

Unfortunately, true randomness is hard to come by. The best source for operating system purposes is to examine hardware processes that are believed to be random in nature, like low order bits of the times required for pieces of hardware to perform operations, and convert the results into random numbers. That's called **gathering entropy**. In Linux, this is done for you automatically, and you can use the gathered entropy by reading `/dev/random`. Windows has a similar entropy-gathering feature. Use these to generate your keys. They're not perfect, but they're good enough for many purposes.

first attempt to add such protection was called WEP (Wired Equivalent Protocol, a rather optimistic name). WEP was constrained by the need to fit into the existing standard, but the method it used to generate and distribute symmetric keys was seriously flawed. Merely by listening in on wireless traffic on an 802.11 network, an attacker could determine the key being used in as little as a minute. There are widely available tools that allow anyone to do so[4].

As another example, an early implementation of the Netscape web browser generated cryptographic keys using some easily guess-able values as seeds to a random number generator, such as the time of day and the ID of the process requesting the key. Researchers discovered they could guess the keys produced in around 30 seconds [GW96].

You might have heard that PK systems use much longer keys, 2K or 4K bits. Sounds much safer, no? Shouldn't that at least make them stronger against brute force attacks? However, you can't select keys for this type of

[4]WEP got replaced by WPA. Unfortunately, WPA proved to have its own weaknesses, so it was replaced by WPA2. Unfortunately, WPA2 proved to have its own weaknesses, so it is being replaced by WPA3, as of 2018. The sad fate of providing cryptography for wireless networks should serve as a lesson to any of you tempted to underestimate the difficulties in getting this stuff right.

cryptosystem at random. Only a relatively few pairs of public and private keys are possible. That's because the public and private keys must be related to each other for the system to work. The relationship is usually mathematical, and usually intended to be mathematically hard to derive, so knowing the public key should not make it easy to learn the private key. However, with the public key in hand, one can use the mathematical properties of the system to derive the private key eventually. That's why PK systems use such big keys – to make sure "eventually" is a very long time.

But that only matters if you keep the private key secret. By now, we hope this sounds obvious, but many makers of embedded devices use PK to provide encryption for those devices, and include a private key in the device's software. All too often, the same private key is used for all devices of a particular model. Such shared private keys invariably become, well, public. In September 2016, one study found 4.5 million embedded devices relying on these private keys that were no longer so private [V16]. Anyone could pose as any of these devices for any purpose, and could read any information sent to them using PK. In essence, the cryptography performed by these devices was little more than window dressing and did not increase the security of the devices by any appreciable amount.

To summarize, cracking cryptography is usually about learning the key. Or, as you might have guessed: **THE CRYPTOGRAPHY'S BENEFIT RELIES ENTIRELY ON THE SECRECY OF THE KEY.**

56.6 Cryptography And Operating Systems

Cryptography is fascinating, but lots of things are fascinating[5], while having no bearing on operating systems. Why did we bother spending half a chapter on cryptography? Because we can use it to protect operating systems.

But not just anywhere and for all purposes. We've pounded into your head that key secrecy is vital for effective use of cryptography. That should make it clear that any time the key can't be kept secret, you can't effectively use cryptography. Casting your mind back to the first chapter on security, remember that the operating system has control of and access to all resources on a computer. Which implies that if you have encrypted information on the computer, and you have the necessary key to decrypt it on the same computer, the operating system on that machine can decrypt the data, whether that was the effect you wanted or not[6].

[5]For example, the late piano Sonatas of Beethoven. One movement of his last Sonata, Opus 111, even sounds like jazz, while being written in the 1820s!

[6]But remember our discussion of security enclaves in an earlier chapter, hardware that does not allow the operating system full access to information that the enclave protects. Think for a moment what the implications of that are for cryptography on a computer using such an enclave, and what new possibilities it offers.

Either you trust your operating system or you don't. If you don't, life is going to be unpleasant anyway, but one implication is that the untrusted operating system, having access at one time to your secret key, can copy it and re-use it whenever it wants to. If, on the other hand, you trust your operating system, you don't need to hide your data from it, so cryptography isn't necessary in this case. This observation has relevance to any situation in which you provide your data to something you don't trust. For instance, if you don't trust your cloud computing facility with your data, you won't improve the situation by giving them your data in plaintext and asking them to encrypt it. They've seen the plaintext and can keep a copy of the key.

If you're sure your operating system is trustworthy right now, but are concerned it might not be later, you can encrypt something now and make sure the key is not stored on the machine. Of course, if you're wrong about the current security of the operating system, or if you ever decrypt the data on the machine after the OS goes rogue, your cryptography will not protect you, since that ever-so-vital secrecy of the key will be compromised.

One can argue that not all compromises of an operating system are permanent. Many are, but some only give an attacker temporary access to system resources, or perhaps access to only a few particular resources. In such cases, if the encrypted data is not stored in plaintext and the decryption key is not available at the time or in the place the attacker can access, encrypting that data may still provide benefit. The tricky issue here is that you can't know ahead of time whether successful attacks on your system will only occur at particular times, for particular durations, or on particular elements of the system. So if you take this approach, you want to minimize all your exposure: decrypt infrequently, dispose of plaintext data quickly and carefully, and don't keep a plaintext version of the key in the system except when performing the cryptographic operations. Such minimization can be difficult to achieve.

If cryptography won't protect us completely against a dishonest operating system, what OS uses for cryptography are there? We saw a specialized example in the chapter on authentication. Some cryptographic operations are one-way: they can encrypt, but never decrypt. We can use these to securely store passwords in encrypted form, even if the OS is compromised, since the encrypted passwords can't be decrypted[7].

What else? In a distributed environment, if we encrypt data on one machine and then send it across the network, all the intermediate components won't be part of our machine, and thus won't have access to the key. The data will be protected in transit. Of course, our partner on the

[7]But if the legitimate user ever provides the correct password to a compromised OS, all bets are off, alas. The compromised OS will copy the password provided by the user and hand it off to whatever villain is working behind the scenes, before it runs the password through the one-way cryptographic hashing algorithm.

final destination machine will need the key if he or she is to use the data. As we promised before, we'll get to that issue in another chapter.

Anything else? Well, what if someone can get access to some of our hardware without going through our operating system? If the data stored on that hardware is encrypted, and the key isn't on that hardware itself, the cryptography will protect the data. This form of encryption is sometimes called **at-rest data encryption**, to distinguish it from encrypting data we're sending between machines. It's useful and important, so let's examine it in more detail.

56.7 At-Rest Data Encryption

As we saw in the chapters on persistence, data can be stored on a disk drive, flash drive, or other medium. If it's sensitive data, we might want some of our desirable security properties, such as secrecy or integrity, to be applied to it. One technique to achieve these goals for this data is to store it in encrypted form, rather than in plaintext. Of course, encrypted data cannot be used in most computations, so if the machine where it is stored needs to perform a general computation on the data, it must first be decrypted[8]. If the purpose is merely to preserve a safe copy of the data, rather than to use it, decryption may not be necessary, but that is not the common case.

The data can be encrypted in different ways, using different ciphers (DES, AES, Blowfish), at different granularities (records, data blocks, individual files, entire file systems), by different system components (applications, libraries, file systems, device drivers). One common general use of at-rest data encryption is called **full disk encryption**. This usually means that the entire contents (or almost the entire contents) of the storage device are encrypted. Despite the name, full-disk encryption can actually be used on many kinds of persistent storage media, not just hard disk drives. Full disk encryption is usually provided either in hardware (built into the storage device) or by system software (a device driver or some element of a file system). In either case, the operating system plays a role in the protection provided. Windows BitLocker and Apple's FileVault are examples of software-based full disk encryption.

Generally, at boot time either the decryption key or information usable to obtain that key (such as a passphrase – like a password, but possibly multiple words) is requested from the user. If the right information is provided, the key or keys necessary to perform the decryption become available (either to the hardware or the operating system). As data is placed on the device, it is encrypted. As data moves off the device, it is

[8] There's one possible exception worth mentioning. Those cryptographic wizards have created a form of cryptography called **homomorphic cryptography**, which allows you to perform operations on the encrypted form of the data without decrypting it. For example, you could add one to an encrypted integer without decrypting it first. When you decrypted the result, sure enough, one would have been added to the original number. Homomorphic ciphers have been developed, but high computational and storage costs render them impractical for most purposes, as of the writing of this chapter. Perhaps that will change, with time.

decrypted. The data remains decrypted as long as it is stored anywhere in the machine's memory, including in shared buffers or user address space. When new data is to be sent to the device, it is first encrypted. The data is never placed on the storage device in decrypted form. After the initial request to obtain the decryption key is performed, encryption and decryption are totally transparent to users and applications. They never see the data in encrypted form and are not asked for the key again, until the machine reboots.

Cryptography is a computationally expensive operation, particularly if performed in software. There will be overhead associated with performing software-based full disk encryption. Reports of the amount of overhead vary, but a few percent extra latency for disk-heavy operations is common. For operations making less use of the disk, the overhead may be imperceptible. For hardware-based full disk encryption, the rated speed of the disk drive will be achieved, which may or may not be slower than a similar model not using full disk encryption.

What does this form of encryption protect against?

- It offers no extra protection against users trying to access data they should not be allowed to see. Either the standard access control mechanisms that the operating system provides work (and such users can't get to the data because they lack access permissions) or they don't (in which case such users will be given equal use of the decryption key as anyone else).
- It does not protect against flaws in applications that divulge data. Such flaws will permit attackers to pose as the user, so if the user can access the unencrypted data, so can the attacker. For example, it offers little protection against buffer overflows or SQL injections.
- It does not protect against dishonest privileged users on the system, such as a system administrator. Administrator's privileges may allow the admin to pose as the user who owns the data or to install system components that provide access to the user's data; thus, the admin could access decrypted copies of the data on request.
- It does not protect against security flaws in the OS itself. Once the key is provided, it is available (directly in memory, or indirectly by asking the hardware to use it) to the operating system, whether that OS is trustworthy and secure or compromised and insecure.

So what benefit does this form of encryption provide? Consider this situation. If a hardware device storing data is physically moved from one machine to another, the OS on the other machine is not obligated to honor the access control information stored on the device. In fact, it need not even use the same file system to access that device. For example, it can treat the device as merely a source of raw data blocks, rather than an organized file system. So any access control information associated with files on the device might be ignored by the new operating system.

However, if the data on the device is encrypted via full disk encryption, the new machine will usually be unable to obtain the encryption key. It can access the raw blocks, but they are encrypted and cannot be decrypted without the key. This benefit would be useful if the hardware in question was stolen and moved to another machine, for example. This situation is a very real possibility for mobile devices, which are frequently lost or stolen. Disk drives are sometimes resold, and data belonging to the former owner (including quite sensitive data) has been found on them by the re-purchaser. These are important cases where full disk encryption provides real benefits.

For other forms of encryption of data at rest, the system must still address the issues of how much is encrypted, how to obtain the key, and when to encrypt and decrypt the data, with different types of protection resulting depending on how these questions are addressed. Generally, such situations require that some software ensures that the unencrypted form of the data is no longer stored anywhere, including caches, and that the cryptographic key is not available to those who might try to illicitly access the data. There are relatively few circumstances where such protection is of value, but there are a few common examples:

- Archiving data that might need to be copied and must be preserved, but need not be used. In this case, the data can be encrypted at the time of its creation, and perhaps never decrypted, or only decrypted under special circumstances under the control of the data's owner. If the machine was uncompromised when the data was first encrypted and the key is not permanently stored on the system, the encrypted data is fairly safe. Note, however, that if the key is lost, you will never be able to decrypt the archived data.
- Storing sensitive data in a cloud computing facility, a variant of the previous example. If one does not completely trust the cloud computing provider (or one is uncertain of how careful that provider is – remember, when you trust another computing element, you're trusting not only its honesty, but also its carefulness and correctness), encrypting the data before sending it to the cloud facility is wise. Many cloud backup products include this capability. In this case, the cryptography and key use occur before moving the data to the untrusted system, or after it is recovered from that system.
- User-level encryption performed through an application. For example, a user might choose to encrypt an email message, with any stored version of it being in encrypted form. In this case, the cryptography will be performed by the application, and the user will do something to make a cryptographic key available to the application. Ideally, that application will ensure that the unencrypted form of the data and the key used to encrypt it are no longer readily available after encryption is completed. Remember, however, that while the key exists, the operating system can obtain access to it without your application knowing.

One important special case for encrypting selected data at rest is a **password vault** (also known as a **key ring**), which we discussed in the authentication chapter. Typical users interact with many remote sites that require them to provide passwords (authentication based on "what you know", remember?) The best security is achieved if one uses a different password for each site, but doing so places a burden on the human user, who generally has a hard time remembering many passwords. A solution is to encrypt all the different passwords and store them on the machine, indexed by the site they are used for. When one of the passwords is required, it is decrypted and provided to the site that requires it.

For password vaults and all such special cases, the system must have some way of obtaining the required key whenever data needs to be encrypted or decrypted. If an attacker can obtain the key, the cryptography becomes useless, so safe storage of the key becomes critical. Typically, if the key is stored in unencrypted form anywhere on the computer in question, the encrypted data is at risk, so well designed encryption systems tend not to do so. For example, in the case of password vaults, the key used to decrypt the passwords is not stored in the machine's stable storage. It is obtained by asking the user for it when required, or asking for a passphrase used to derive the key. The key is then used to decrypt the needed password. Maximum security would suggest destroying the key as soon as this decryption was performed (remember the principle of least privilege?), but doing so would imply that the user would have to re-enter the key each time a password was needed (remember the principle of acceptability?). A compromise between usability and security is reached, in most cases, by remembering the key after first entry for a significant period of time, but only keeping it in RAM. When the user logs out, or the system shuts down, or the application that handles the password vault (such as a web browser) exits, the key is "forgotten." This approach is reminiscent of single sign-on systems, where a user is asked for a password when the system is first accessed, but is not required to re-authenticate again until logging out. It has the same disadvantages as those systems, such as permitting an unattended terminal to be used by unauthorized parties to use someone else's access permissions. Both have the tremendous advantage that they don't annoy their users so much that they are abandoned in favor of systems offering no security whatsoever.

56.8 Cryptographic Capabilities

Remember from our chapter on access control that capabilities had the problem that we could not leave them in users' hands, since then users could forge them and grant themselves access to anything they wanted. Cryptography can be used to create unforgeable capabilities. A trusted entity could use cryptography to create a sufficiently long and securely encrypted data structure that indicated that the possessor was allowed to have access to a particular resource. This data structure could then be given to a user, who would present it to the owner of the matching resource to obtain access. The system that actually controlled the resource must be able to check the validity of the data structure before granting access, but would not need to maintain an access control list.

Such cryptographic capabilities could be created either with symmetric or public key cryptography. With symmetric cryptography, both the creator of the capability and the system checking it would need to share the same key. This option is most feasible when both of those entities are the same system, since otherwise it requires moving keys around between the machines that need to use the keys, possibly at high speed and scale, depending on the use scenario. One might wonder why the single machine would bother creating a cryptographic capability to allow access, rather than simply remembering that the user had passed an access check, but there are several possible reasons. For example, if the machine controlling the resource worked with vast numbers of users, keeping track of the access status for each of them would be costly and complex, particularly in a distributed environment where the system needed to worry about failures and delays. Or if the system wished to give transferable rights to the access, as it might if the principal might move from machine to machine, it would be more feasible to allow the capability to move with the principal and be used from any location. Symmetric cryptographic capabilities also make sense when all of the machines creating and checking them are inherently trusted and key distribution is not problematic.

If public key cryptography is used to create the capabilities, then the creator and the resource controller need not be co-located and the trust relationships need not be as strong. The creator of the capability needs one key (typically the secret key) and the controller of the resource needs the other. If the content of the capability is not itself secret, then a true public key can be used, with no concern over who knows it. If secrecy (or at least some degree of obscurity) is required, what would otherwise be a public key can be distributed only to the limited set of entities that would need to check the capabilities[9]. A resource manager could create a set of credentials (indicating which principal was allowed to use what resources, in what ways, for what period of time) and then encrypt them with a private key. Any one else can validate those credentials by decrypting them with the manager's public key. As long as only the resource manager knows the private key, no one can forge capabilities.

As suggested above, such cryptographic capabilities can hold a good deal of information, including expiration times, identity of the party who was given the capability, and much else. Since strong cryptography will ensure integrity of all such information, the capability can be relied upon. This feature allows the creator of the capability to prevent arbitrary copying and sharing of the capability, at least to a certain extent. For example, a cryptographic capability used in a network context can be tied to a particular IP address, and would only be regarded as valid if the message carrying it came from that address.

[9]Remember, however, that if you are embedding a key in a piece of widely distributed software, you can count on that key becoming public knowledge. So even if you believe the matching key is secret, not public, it is unwise to rely too heavily on that belief.

Many different encryption schemes can be used. The important point is that the encrypted capabilities must be long enough that it is computationally infeasible to find a valid capability by brute force enumeration or random guessing (e.g., the number of invalid bit patterns is 10^{15} times larger than the number of valid bit patterns).

We'll say a bit more about cryptographic capabilities in the chapter on distributed system security.

56.9 Summary

Cryptography can offer certain forms of protection for data even when that data is no longer in a system's custody. These forms of protection include secrecy, integrity, and authentication. Cryptography achieves such protection by converting the data's original bit pattern into a different bit pattern, using an algorithm called a cipher. In most cases, the transformation can be reversed to obtain the original bit pattern. Symmetric ciphers use a single secret key shared by all parties with rights to access the data. Asymmetric ciphers use one key to encrypt the data and a second key to decrypt the data, with one of the keys kept secret and the other commonly made public. Cryptographic hashes, on the other hand, do not allow reversal of the cryptography and do not require the use of keys.

Strong ciphers make it computationally infeasible to obtain the original bit pattern without access to the required key. For symmetric and asymmetric ciphers, this implies that only holders of the proper key can obtain the cipher's benefits. Since cryptographic hashes have no key, this implies that no one should be able to obtain the original bit pattern from the hash.

For operating systems, the obvious situations in which cryptography can be helpful are when data is sent to another machine, or when hardware used to store the data might be accessed without the intervention of the operating system. In the latter case, data can be encrypted on the device (using either hardware or software), and decrypted as it is delivered to the operating system.

Ciphers are generally not secret, but rather are widely known and studied standards. A cipher's ability to protect data thus relies entirely on key secrecy. If attackers can learn, deduce, or guess the key, all protection is lost. Thus, extreme care in key selection and maintaining key secrecy is required if one relies on cryptography for protection. A good principle is to store keys in as few places as possible, for as short a duration as possible, available to as few parties as possible.

References

[D88] "The First Ten Years of Public Key Cryptography" by Whitfield Diffie. Communications of the ACM, Vol. 76, No. 5, May 1988. *A description of the complex history of where public key cryptography came from.*

[D+14] "The Matter of Heartbleed" by Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. Proceedings of the 2014 Conference on Internet Measurement Conference. *A good description of the Heartbleed vulnerability in OpenSSL and its impact on the Internet as a whole. Worth reading for the latter, especially, as it points out how one small bug in one critical piece of system software can have a tremendous impact.*

[G02] "Lessons Learned in Implementing and Deploying Crypto Software" by Peter Gutmann. Usenix Security Symposium, 2002. *A good analysis of the many ways in which poor use of a perfectly good cipher can totally compromise your software, backed up by actual cases of the problems occurring in the real world.*

[G17] "SHA-1 Shattered" by Google. `https://shattered.io`, 2017. *A web site describing details of how Google demonstrated the insecurity of the SHA-1 cryptographic hashing function. The web site provides general details, but also includes a link to a technical paper describing exactly how it was done.*

[GW96] "Randomness and the Netscape Browser" by Ian Goldberg and David Wagner. Dr. Dobbs Journal, January 1996. *Another example of being able to deduce keys that were not properly created and handled, in this case by guessing the inputs to the random number generator used to create the keys. Aren't attackers clever? Don't you wish they weren't?*

[K96] "The Codebreakers" by David Kahn. Scribner Publishing, 1996. *A long, but readable, history of cryptography, its uses, and how it is attacked.*

[S96] "Applied Cryptography" by Bruce Schneier. Jon Wiley and Sons, Inc., 1996. *A detailed description of how to use cryptography in many different circumstances, including example source code.*

[V16] "House of Keys: 9 Months later... 40% Worse" by Stefan Viehbock. Available on: `blog.sec-consult.com/2016/09/house-of-keys-9-months-later-40-worse.html`. *A web page describing the unfortunate ubiquity of the same private key being used in many different embedded devices.*

57

Distributed System Security

Chapter by **Peter Reiher (UCLA)**

57.1 Introduction

An operating system can only control its own machine's resources. Thus, operating systems will have challenges in providing security in distributed systems, where more than one machine must cooperate. There are two large problems:

- The other machines in the distributed system might not properly implement the security policies you want, or they might be adversaries impersonating trusted partners. We cannot control remote systems, but we still have to be able to trust validity of the credentials and capabilities they give us.
- Machines in a distributed system communicate across a network that none of them fully control and that, generally, cannot be trusted. Adversaries often have equal access to that network and can forge, copy, replay, alter, destroy, and delay our messages, and generally interfere with our attempts to use the network.

As suggested earlier, cryptography will be the major tool we use here, but we also said cryptography was hard to get right. That makes it sound like the perfect place to use carefully designed standard tools, rather than to expect everyone to build their own. That's precisely correct. As such:

> THE CRUX: HOW TO PROTECT DISTRIBUTED SYSTEM OPERATIONS
> How can we secure a system spanning more than one machine? What tools are available to help us protect such systems? How do we use them properly? What are the areas in using the tools that require us to be careful and thoughtful?

57.2 The Role of Authentication

How can we handle our uncertainty about whether our partners in a distributed system are going to enforce our security policies? In most cases, we can't do much. At best, we can try to arrange to agree on policies and hope everyone follows through on those agreements. There are some special cases where we can get high-quality evidence that our partners have behaved properly, but that's not easy, in general. For example, how can we know that they are using full disk encryption, or that they have carefully wiped an encryption key we are finished using, or that they have set access controls on the local copies of their files properly? They can say they did, but how can we *know*?

Generally, we can't. But you're used to that. In the real world, your friends and relatives know some secrets about you, and they might have keys to get into your home, and if you loan them your car you're fairly sure you'll get it back. That's not so much because you have perfect mechanisms to prevent those trusted parties from behaving badly, but because you are pretty sure they won't. If you're wrong, perhaps you can detect that they haven't behaved well and take compensating actions (like changing your locks or calling the police to report your car stolen). We'll need to rely on similar approaches in distributed computer systems. We will simply have to trust that some parties will behave well. In some cases, we can detect when they don't and adjust our trust in the parties accordingly, and maybe take other compensating actions.

Of course, in the cyber world, our actions are at a distance over a network, and all we see are bits going out and coming in on the network. For a trust-based solution to work, we have to be quite sure that the bits we send out can be verified by our buddies as truly coming from us, and we have to be sure that the bits coming in really were created by them. That's a job for authentication. As suggested in the earlier authentication chapter, when working over a network, we need to authenticate based on a bundle of bits. Most commonly, we use a form of authentication based on what you know. Now, think back to the earlier chapters. What might someone running on a remote operating system know that no one else knows? How about a password? How about a private key?

Most of our distributed system authentication will rely on one of these two elements. Either you require the remote machine to provide you with a password, or you require it to provide evidence using a private key stored only on that machine[1]. In each case, you need to know something to check the authentication: either the password (or, better, a cryptographic hash of the password plus a salt) or the public key.

[1] We occasionally use other methods, such as smart cards or remote biometric readers. They are less common in today's systems, though. If you understand how we use passwords and public key cryptography for distributed system authentication, you can probably figure out how to make proper use of these other techniques, too. If you don't, you'll be better off figuring out the common techniques before moving to the less common ones.

When is each appropriate? Passwords tend to be useful if there are a vast number of parties who need to authenticate themselves to one party. Public keys tend to be useful if there's one party who needs to authenticate himself to a vast number of parties. Why? With a password, the authentication provides evidence that somebody knows a password. If you want to know exactly who that is (which is usually important), only the party authenticating and the party checking can know it. With a public key, many parties can know the key, but only one party who knows the matching private key can authenticate himself. So we tend to use both mechanisms, but for different cases. When a web site authenticates itself to a user, it's done with PK cryptography. By distributing one single public key (to vast numbers of users), the web site can be authenticated by all its users. The web site need not bother keeping separate authentication information to authenticate itself to each user. When that user authenticates itself to the web site, it's done with a password. Each user must be separately authenticated to the web site, so we require a unique piece of identifying information for that user, preferably something that's easy for a person to use. Setting up and distributing public keys is hard, while setting up individual passwords is relatively easy.

How, practically, do we use each of these authentication mechanisms in a distributed system? If we want a remote partner to authenticate itself via passwords, we will require it to provide us with that password, which we will check. We'll need to encrypt the transport of the password across the network if we do that; otherwise anyone eavesdropping on the network (which is easy for many wireless networks) will readily learn passwords sent unencrypted. Encrypting the password will require that we already have either a shared symmetric key or our partner's public key. Let's concentrate now on how we get that public key, either to use it directly or set up the cryptography to protect the password in transit.

We'll spend the rest of the chapter on securing the network connection, but please don't forget that even if you secure the network perfectly, you still face the major security challenge of the uncontrolled site you're interacting with on the other side of the network. If your compromised partner attacks you, it will offer little consolation that the attack was authenticated and encrypted.

57.3 Public Key Authentication For Distributed Systems

The public key doesn't need to be secret, but we need to be sure it really belongs to our partner. If we have a face-to-face meeting, our partner can directly give us a public key in some form or another, in which case we can be pretty sure it's the right one. That's limiting, though, since we often interact with partners whom we never see face to face. For that matter, whose "face" belongs to Amazon[2] or Google?

[2] How successful would Amazon be if Jeff Bezos had to make an in-person visit to every customer to deliver them Amazon's public key? Answer: Not as successful.

Fortunately, we can use the fact that secrecy isn't required to simply create a bunch of bits containing the public key. Anyone who gets a copy of the bits has the key. But how do they know for sure whose key it is? What if some other trusted party known to everyone who needs to authenticate our partner used their own public key to cryptographically sign that bunch of bits, verifying that they do indeed belong to our partner? If we could check that signature, we could then be sure that bunch of bits really does represent our partner's public key, at least to the extent that we trust that third party who did the signature.

This technique is how we actually authenticate web sites and many other entities on the Internet. Every time you browse the web or perform any other web-based activity, you use it. The signed bundle of bits is called a **certificate**. Essentially, it contains information about the party that owns the public key, the public key itself, and other information, such as an expiration date. The entire set of information, including the public key, is run through a cryptographic hash, and the result is encrypted with the trusted third party's private key, digitally signing the certificate. If you obtain a copy of the certificate, and can check the signature, you can learn someone else's public key, even if you have never met or had any direct interaction with them. In certain ways, it's a beautiful technology that empowers the whole Internet.

Let's briefly go through an example, to solidify the concepts. Let's say Frobazz Inc. wants to obtain a certificate for its public key, which is KF. Frobazz Inc. pays big bucks to Acmesign Co., a widely trusted company whose business it is to sell certificates, to obtain a certificate signed by AcmeSign. Such companies are commonly called **Certificate Authorities**, or **CAs**, since they create authoritative certificates trusted by many parties. Acmesign checks up on Frobazz Inc. to ensure that the people asking for the certificate actually are legitimate representatives of Frobazz. Acmesign then makes very, very sure that the public key it's about to embed in a certificate actually is the one that Frobazz wants to use. Assuming it is, Acmesign runs a cryptographic hashing algorithm (perhaps SHA-3 which, unlike SHA-1, has not been cracked, as of 2020) on Frobazz's name, public key KF, and other information, producing hash HF. Acmesign then encrypts HF with its own private key, PA, producing digital signature SF. Finally, Acmesign combines all the information used to produce HF, plus Acmesign's own identity and the signature SF, into the certificate CF, which it hands over to Frobazz, presumably in exchange for money. Remember, CF is just some bits.

Now Frobazz Inc. wants to authenticate itself over the Internet to one of its customers. If the customer already has Frobazz's public key, we can use public key authentication mechanisms directly. If the customer does not have the public key, Frobazz sends CF to the customer. The customer examines the certificate, sees that it was generated by Acmesign using, say, SHA-3, and runs the same information that Acmesign hashed (all of which is in the certificate itself) through SHA-3, producing HF'. Then the customer uses Acmesign's public key to decrypt SF (also in the certificate), obtaining HF. If all is well, HF equals HF', and now the customer knows that the public key in the certificate is indeed Frobazz's. Public key-based authentication can proceed[3]. If the two hashes aren't exactly the same, the customer knows that something fishy is going on and will not accept the certificate.

There are some wonderful properties about this approach to learning public keys. First, note that the signing authority (Acmesign, in our example) did not need to participate in the process of the customer checking the certificate. In fact, Frobazz didn't really, either. The customer can get the certificate from literally anywhere and obtain the same degree of assurance of its validity. Second, it only needs to be done once per customer. After obtaining the certificate and checking it, the customer has the public key that is needed. From that point onward, the customer can simply store it and use it. If, for whatever reason, it gets lost, the customer can either extract it again from the certificate (if that has been saved), or go through the process of obtaining the certificate again. Third, the customer had no need to trust the party claiming to be Frobazz until that identity had been proven by checking the certificate. The customer can proceed with caution until the certificate checks out.

Assuming you've been paying attention for the last few chapters, you should be saying to yourself, "now, wait a minute, isn't there a chicken-and-egg problem here?" We'll learn Frobazz's public key by getting a certificate for it. The certificate will be signed by Acmesign. We'll check the signature by knowing Acmesign's public key. But where did we get Acmesign's key? We really hope you did have that head-scratching moment and asked yourself that question, because if you did, you understand the true nature of the Internet authentication problem. Ultimately, we've got to bootstrap it. You've got to somehow or other obtain a public key for somebody that you trust. Once you do, if it's the right public key for the right kind of party, you can then obtain a lot of other public keys. But without something to start from, you can't do much of anything.

Where do you get that primal public key? Most commonly, it comes in a piece of software you obtain and install. The one you use most often is probably your browser, which typically comes with the public keys for several hundred trusted authorities[4]. Whenever you go to a new web site that cares about security, it provides you with a certificate containing that site's public key, and signed by one of those trusted authorities pre-configured into your browser. You use the pre-configured public key of that authority to verify that the certificate is indeed proper, after which you know the public key of that web site. From that point onward, you can use the web site's public key to authenticate it. There are some se-

[3] And, indeed, must, since all this business with checking the certificate merely told the customer what Frobazz's public key was. It did nothing to assure the customer that whoever sent the certificate actually was Frobazz or knew Frobazz's private key.

[4] You do know of several hundred companies out there that you trust with everything you do on the web, don't you? Well, know of them or not, you effectively trust them to that extent.

rious caveats here (and some interesting approaches to addressing those caveats), but let's put those aside for the moment.

Anyone can create a certificate, not just those trusted CAs, either by getting one from someone whose business it is to issue certificates or simply by creating one from scratch, following a certificate standard (X.509 is the most commonly used certificate standard [I12]). The necessary requirement: the party being authenticated and the parties performing the authentication must all trust whoever created the certificate. If they don't trust that party, why would they believe the certificate is correct?

If you are building your own distributed system, you can create your own certificates from a machine you (and other participants in the system) trust and can handle the bootstrapping issue by carefully hand-installing the certificate signing machine's public key wherever it needs to be. There are a number of existing software packages for creating certificates, and, as usual with critical cryptographic software, you're better off using an existing, trusted implementation rather than coding up one of your own. One example you might want to look at is PGP (available in both supported commercial versions and compatible but less supported free versions) [P16], but there are others. If you are working with a fixed number of machines and you can distribute the public key by hand in some reasonable way, you can dispense entirely with certificates. Remember, the only point of a PK certificate is to distribute the public key, so if your public keys are already where they need to be, you don't need certificates.

OK, one way or another you've obtained the public key you need to authenticate some remote machine. Now what? Well, anything they send you encrypted with their private key will only decrypt with their public key, so anything that decrypts properly with the public key must have come from them, right? Yes, it must have come from them at some point, but it's possible for an adversary to have made a copy of a legitimate message the site sent at some point in the past and then send it again it at some future date. Depending on exactly what's going on, that could cause trouble, since you may take actions based on that message that the legitimate site did not ask for. So usually we take measures to ensure that we're not being subjected to a **replay attack**. Such measures generally involve ensuring that each encrypted message contains unique information not in any other message. This feature is built in properly to standard cryptographic protocols, so if you follow our advice and use one of those, you will get protection from such replay attacks. If you insist on building your own cryptography, you'll need to learn a good deal more about this issue and will have to apply that knowledge very carefully. Also, public key cryptography is expensive. We want to stop using it as soon as possible, but we also want to continue to get authentication guarantees. We'll see how to do that when we discuss SSL and TLS.

57.4 Password Authentication For Distributed Systems

The other common option to authenticate in distributed systems is to use a password. As noted above, that will work best in situations where only two parties need to deal with any particular password: the party being authenticated and the authenticating party. They make sense when an individual user is authenticating himself to a site that hosts many users, such as when you log in to Amazon. They don't make sense when that site is trying to authenticate itself to an individual user, such as when a web site claiming to be Amazon wants to do business with you. Public key authentication works better there.

How do we properly handle password authentication over the network, when it is a reasonable choice? The password is usually associated with a particular user ID, so the user provides that ID and password to the site requiring authentication. That typically happens over a network, and typically we cannot guarantee that networks provide confidentiality. If our password is divulged to someone else, they'll be able to pose as us, so we must add confidentiality to this cross-network authentication, generally by encrypting at least the password itself (though encrypting everything involved is better). So a typical interchange with Alice trying to authenticate herself to Frobazz Inc.'s web site would involve the site requesting a user ID and password and Alice providing both, but encrypting them before sending them over the network.

The obvious question you should ask is, encrypting them with what key? Well, if Frobazz authenticated itself to Alice using PK, as discussed above, Alice can encrypt her user ID and password with Frobazz's public key. Frobazz Inc., having the matching private key, will be able to check them, but nobody else can read them. In actuality, there are various reasons why this alone would not suffice, including replay attacks, as mentioned above. But we can and do use Frobazz's private key to set up cryptography that will protect Alice's password in transit. We'll discuss the details in the section on SSL/TLS.

We discussed issues of password choice and management in the chapter on authentication, and those all apply in the networking context. Otherwise, there's not that much more to say about how we'll use passwords, other than to note that after the remote site has verified the password, what does it actually know? That the site or user who sent the password knows it, and, to the strength of the password, that site or user is who it claims to be. But what about future messages that come in, supposedly from that site? Remember, anyone can create any message they want, so if all we do is verify that the remote site sent us the right password, all we know is that particular message is authentic. We don't want to have to include the password on every message we send, just as we don't want to use PK to encrypt every message we send. We will use both authentication techniques to establish initial authenticity, then use something else to tie that initial authenticity to subsequent interactions. Let's move right along to SSL/TLS to talk about how we do that.

57.5 SSL/TLS

We saw in an earlier chapter that a standard method of communicating between processes in modern systems is the socket. That's equally true when the processes are on different machines. So a natural way to add cryptographic protection to communications crossing unprotected networks is to add cryptographic features to sockets. That's precisely what **SSL** (the **Secure Socket Layer**) was designed to do, many years ago. Unfortunately, SSL did not get it quite right. That's because it's pretty darn hard to get it right, not because the people who designed and built it were careless. They learned from their mistakes and created a new version of encrypted sockets called **Transport Layer Security** (**TLS**)[5]. You will frequently hear people talk about using SSL. They are usually treating it as a shorthand for SSL/TLS. SSL, formally, is insecure and should never be used for anything. Use TLS. The only exception is that some very old devices might run software that doesn't support TLS. In that case, it's better to use SSL than nothing. We'll adopt the same shorthand as others from here on, since it's ubiquitous.

The concept behind SSL is simple: move encrypted data through an ordinary socket. You set up a socket, set up a special structure to perform whatever cryptography you want, and hook the output of that structure to the input of the socket. You reverse the process on the other end. What's simple in concept is rather laborious in execution, with a number of steps required to achieve the desired result. There are further complications due to the general nature of SSL. The technology is designed to support a variety of cryptographic operations and many different ciphers, as well as multiple methods to perform key exchange and authentication between the sender and receiver.

The process of adding SSL to your program is intricate, requiring the use of particular libraries and a sequence of calls into those libraries to set up a correct SSL connection. We will not go through those operations step by step here, but you will need to learn about them to make proper use of SSL. Their purpose is, for the most part, to allow a wide range of generality both in the cryptographic options SSL supports and the ways you use those options in your program. For example, these setup calls would allow you to create one set of SSL connections using AES-128 and another using AES-256, if that's what you needed to do.

One common requirement for setting up an SSL connection that we will go through in a bit more detail is how to securely distribute whatever cryptographic key you will use for the connection you are setting up. Best cryptographic practice calls for you to use a brand new key to encrypt the bulk of your data for each connection you set up. You will use

[5] Actually, even the first couple of versions of TLS didn't get it quite right. As of 2020, the current version of TLS is 1.3, and that's probably what you should use. TLS 1.3 closed some vulnerabilities that TLS 1.2 is subject to, The history of required changes to SSL/TLS should further reinforce the lesson of how hard it is to use cryptography properly, which in turn should motivate you to foreswear ever trying to roll your own crypto.

public/private keys for authentication many times, but as we discussed earlier, you need to use symmetric cryptography to encrypt the data once you have authenticated your partner, and you want a fresh key for that. Even if you are running multiple simultaneous SSL connections with the same partner, you want a different symmetric key for each connection.

So what do you need to do to set up a new SSL connection? We won't go through all of the gory details, but, in essence, SSL needs to bootstrap a secure connection based (usually) on asymmetric cryptography when no usable symmetric key exists. (You'll hear "usually" and "normally" and "by default" a lot in SSL discussions, because of SSL's ability to support a very wide range of options, most of which are ordinarily not what you want to do.) The very first step is to start a negotiation between the client and the server. Each party might only be able to handle particular ciphers, secure hashes, key distribution strategies, or authentication schemes, based on what version of SSL they have installed, how it's configured, and how the programs that set up the SSL connection on each side were written. In the most common cases, the negotiation will end in both sides finding some acceptable set of ciphers and techniques that hit a balance between security and performance. For example, they might use RSA with 2048 bit keys for asymmetric cryptography, some form of a Diffie-Hellman key exchange mechanism (see the Aside on this mechanism) to establish a new symmetric key, SHA-3 to generate secure hashes for integrity, and AES with 256 bit keys for bulk encryption. A modern installation of SSL might support 50 or more different combinations of these options.

In some cases, it may be important for you to specify which of these many combinations are acceptable for your system, but often most of them will do, in which case you can let SSL figure out which to use for each connection without worrying about it yourself. The negotiation will happen invisibly and SSL will get on with its main business: authenticating at least the server (optionally the client), creating and distributing a new symmetric key, and running the communication through the chosen cipher using that key.

We can use Diffie-Hellman key exchange to create the key (and SSL frequently does), but we need to be sure who we are sharing that key with. SSL offers a number of possibilities for doing so. The most common method is for the client to obtain a certificate containing the server's public key (typically by having the server send it to the client) and to use the public key in that certificate to verify the authenticity of the server's messages. It is possible for the client to obtain the certificate through some other means, though less common. Note that having the server send the certificate is every bit as secure (or insecure) as having the client obtain the certificate through other means. Certificate security is not based on the method used to transport it, but on the cryptography embedded in the certificate.

With the certificate in hand (however the client got it), the Diffie-Hellman key exchange can now proceed in an authenticated fashion. The server

ASIDE: DIFFIE-HELLMAN KEY EXCHANGE

What if you want to share a secret key between two parties, but they can only communicate over an insecure channel, where eavesdroppers can hear anything they say? You might think this is an impossible problem to solve, but you'd be wrong. Two extremely smart cryptographers named Whitfield Diffie and Martin Hellman solved this problem years ago, and their solution is in common use. It's called **Diffie-Hellman key exchange**.

Here's how it works. Let's say Alice and Bob want to share a secret key, but currently don't share anything, other than the ability to send each other messages. First, they agree on two numbers, n (a large prime number) and g (which is primitive mod n). They can use the insecure channel to do this, since n and g don't need to be secret. Alice chooses a large random integer, say x, calculates $X = g^x mod\ n$, and sends X to Bob. Bob independently chooses a large random integer, say y, calculates $Y = g^y mod\ n$, and sends Y to Alice. The eavesdroppers can hear X and Y, but since Alice and Bob didn't send x or y, the eavesdroppers don't know those values. It's important that Alice and Bob keep x and y secret.

Alice now computes $k = Y^x mod\ n$, and Bob computes $k = X^y mod\ n$. Alice and Bob get the same value k from these computations. Why? Well, $Y^x mod\ n = (g^y\ mod\ n)^x\ mod\ n$, which in turn equals $g^{yx}\ mod\ n$. $X^y\ mod\ n = (g^x\ mod\ n)^y\ mod\ n = g^{xy}\ mod\ n$, which is the same thing Alice got. Nothing magic there, that's just how exponentiation and modulus arithmetic work. Ah, the glory of mathematics! So k is the same in both calculations and is known to both Alice and Bob.

What about those eavesdroppers? They know g, n, X, and Y, but not x or y. They can compute $k' = XY\ mod\ n$, but that is not equal to the k Alice and Bob calculated. They do have approaches to derive x or y, which would give them enough information to obtain k, but those approaches require them either to perform a calculation for every possible value of n (which is why you want n to be very large) or to compute a discrete logarithm. Computing a discrete logarithm is a solvable problem, but it's computationally infeasible for large numbers. So if the prime n is large (and meets other properties), the eavesdroppers are out of luck. How large? 600 digit primes should be good enough.

Neat, no? But there is a fly in the ointment, when one considers using Diffie-Hellman over a network. It ensures that you securely share a key with someone, but gives you no assurance of who you're sharing the key with. Maybe Alice is sharing the key with Bob, as she thinks and hopes, but maybe she's sharing it with Mallory, who posed as Bob and injected his own Y. Since we usually care who we're in secure communication with, we typically augment Diffie-Hellman with an authentication mechanism to provide the assurance of our partner's identity.

will sign its Diffie-Hellman messages with its private key, which will allow the client to determine that its partner in this key exchange is the correct server. Typically, the client does not provide (or even have) its own certificate, so it cannot sign its Diffie-Hellman messages. This implies that when SSL's Diffie-Hellman key exchange completes, typically the client is pretty sure who the server is, but the server has no clue about the client's identity. (Again, this need not be the case for all uses of SSL. SSL includes connection creation options where both parties know each other's public key and the key exchange is authenticated on both sides. Those options are simply not the most commonly used ones, and particularly are not the ones typically used to secure web browsing.)

Recalling our discussion earlier in this chapter, it actually isn't a problem for the server to be unsure about the client's identity at this point, in many cases. As we stated earlier, the client will probably want to use a password to authenticate itself, not a public key extracted from a certificate. As long as the server doesn't permit the client to do anything requiring trust before the server obtains and checks the client's password, the server probably doesn't care who the client is, anyway. Many servers offer some services to anonymous clients (such as providing them with publicly available information), so as long as they can get a password from the client before proceeding to more sensitive subjects, there is no security problem. The server can ask the client for a user ID and password later, at any point after the SSL connection is established. Since creating the SSL connection sets up a symmetric key, the exchange of ID and password can be protected with that key.

A final word about SSL/TLS: it's a protocol, not a software package. There are multiple different software packages that implement this protocol. Ideally, if they all implement the protocol properly, they all interact correctly. However, they use different code to implement the protocol. As a result, software flaws in one implementation of SSL/TLS might not be present in other implementations. For example, the Heartbleed attack was based on implementation details of OpenSSL [H14], but was not present in other implementations, such as the version of SSL/TLS found in Microsoft's Windows operating system. It is also possible that the current protocol definition of SSL/TLS contains protocol flaws that would be present in any compliant implementation. If you hear of a security problem involving SSL, determine whether it is a protocol flaw or an implementation flaw before taking further action. If it's an implementation flaw, and you use a different implementation, you might not need to take any action in response.

57.6 Other Authentication Approaches

While passwords and public keys are the most common ways to authenticate a remote user or machines, there are other options. One such option is used all the time. After you have authenticated yourself to a web site by providing a password, as we described above, the web site will continue to assume that the authentication is valid. It won't ask for your password every time you click a link or perform some other interaction with it. (And a good thing, too. Imagine how much of a pain it would be if you had to provide your password every time you wanted to do anything.) If your session is encrypted at this point, it could regard your proper use of the cryptography as a form of authentication; but you might even be able to quit your web browser, start it up again, navigate back to that web site, and still be treated as an authenticated user, without a new request for your password. At that point, you're no longer using the same cryptography you used before, since you would have established a new session and set up a new cryptographic key. How did your partner authenticate that you were the one receiving the new key?

In such cases, the site you are working with has chosen to make a security tradeoff. It verified your identity at some time in the past using your password and then relies on another method to authenticate you in the future. A common method is to use **web cookies**. Web cookies are pieces of data that a web site sends to a client with the intention that the client stores that data and send it back again whenever the client next communicates with the server. Web cookies are built into most browsers and are handled invisibly, without any user intervention. With proper use of cryptography, a server that has verified the password of a client can create a web cookie that securely stores the client's identity. When the client communicates with the server again, the web browser automatically includes the cookie in the request, which allows the server to verify the client's identity without asking for a password again[6].

If you spend a few minutes thinking about this authentication approach, you might come up with some possible security problems associated with it. The people designing this technology have dealt with some of these problems, like preventing an eavesdropper from simply using a cookie that was copied as it went across the network. However, there are other security problems (like someone other than the legitimate user using the computer that was running the web browser and storing the cookie) that can't be solved with these kinds of cookies, but could have been solved if you required the user to provide the password every time. When you build your own system, you will need to think about these sorts of security tradeoffs yourself. Is it better to make life simpler for your user by not asking for a password except when absolutely necessary, or is it better to provide your user with improved security by frequently requiring proof of identity? The point isn't that there is one correct an-

[6]You might remember from the chapter on access control that we promised to discuss protecting capabilities in a network context using cryptography. That, in essence, is what these web cookies are. After a user authenticates itself with another mechanism, the remote system creates a cryptographic capability for that user that no one else could create, generally using a key known only to that system. That capability/cookie can now be passed back to the other party and used for future authorization operations. The same basic approach is used in a lot of other distributed systems.

swer to this question, but that you need to think about such questions in the design of your system.

There are other authentication options. One example is what is called a **challenge/response protocol**. The remote machine sends you a challenge, typically in the form of a number. To authenticate yourself, you must perform some operation on the challenge that produces a response. This should be an operation that only the authentic party can perform, so it probably relies on the use of a secret that party knows, but no one else does. The secret is applied to the challenge, producing the response, which is sent to the server. The server must be able to verify that the proper response has been provided. A different challenge is sent every time, requiring a different response, so attackers gain no advantage by listening to and copying down old challenges and responses. Thus, the challenges and responses need not be encrypted. Challenge/response systems usually perform some kind of cryptographic operation, perhaps a hashing operation, on the challenge plus the secret to produce the response. Such operations are better performed by machines than people, so either your computer calculates the response for you or you have a special hardware token that takes care of it. Either way, a challenge/response system requires pre-arrangement between the challenging machine and the machine trying to authenticate itself. The hardware token or data secret must have been set up and distributed before the challenge is issued.

Another authentication option is to use an authentication server. In essence, you talk to a server that you trust and that trusts you. The party you wish to authenticate to must also trust the server. The authentication server vouches for your identity in some secure form, usually involving cryptography. The party who needs to authenticate you is able to check the secure information provided by the authentication server and thus determine that the server verified your identity. Since the party you wish to communicate with trusts the authentication server, it now trusts that you are who you claim to be. In a vague sense, certificates and CAs are an offline version of such authentication servers. There are more active online versions that involve network interactions of various sorts between the two machines wishing to communicate and one or more authentication servers. Online versions are more responsive to changes in security conditions than offline versions like CAs. An old certificate that should not be honored is hard to get rid of, but an online authentication server can invalidate authentication for a compromised party instantly and apply the changes immediately. The details of such systems can be quite complex, so we will not discuss them in depth. Kerberos is one example of such an online authentication server [NT95].

57.7 Some Higher Level Tools

In some cases, we can achieve desirable security effects by working at a higher level. **HTTPS** (the cryptographically protected version of the HTTP protocol) and **SSH** (a competitor to SSL most often used to set up secure sessions with remote computers) are two good examples.

HTTPS

HTTP, the protocol that supports the World Wide Web, does not have its own security features. Nowadays, though, much sensitive and valuable information is moved over the web, so sending it all unprotected over the network is clearly a bad idea. Rather than come up with a fresh implementation of security for HTTP, however, HTTPS takes the existing HTTP definition and connects it to SSL/TLS. SSL takes care of establishing a secure connection, including authenticating the web server using the certificate approach discussed earlier and establishing a new symmetric encryption key known only to the client and server. Once the SSL connection is established, all subsequent interactions between the client and server use the secured connection. To a large extent, HTTPS is simply HTTP passed through an SSL connection.

That does not devalue the importance of HTTPS, however. In fact, it is a useful object lesson. Rather than spend years in development and face the possibility of the same kinds of security flaws that other developers of security protocols inevitably find, HTTPS makes direct use of a high quality transport security tool, thus replacing an insecure transport with a highly secure transport at very little development cost.

HTTPS obviously depends heavily on authentication, since we want to be sure we aren't communicating with malicious web sites. HTTPS uses certificates for that purpose. Since HTTPS is intended primarily for use in web browsers, the certificates in question are gathered and managed by the browser. Modern browsers come configured with the public keys of many certificate signing authorities (CAs, as we mentioned earlier). Certificates for web sites are checked against these signing authorities to determine if the certificate is real or bogus. Remember, however, what a certificate actually tells you, assuming it checks out: that at some moment in time the signing authority thoughts it was a good idea to vouch that a particular public key belongs to a particular party. There is no implication that the party is good or evil, that the matching private key is still secret, or even that the certificate signing authority itself is secure and uncompromised, either when it created the certificate or at the moment you check it. There have been real world problems with web certificates for all these reasons. Remember also that HTTPS only vouches for authenticity. An authenticated web site using HTTPS can still launch an attack on your client. An authenticated attack, true, but that won't be much consolation if it succeeds.

Not all web browsers always supported HTTPS, typically because they didn't have SSL installed or configured. In those cases, a web site using HTTPS only would not be able to interact with the client, since the client couldn't set up its end of the SSL socket. The standard solution for web servers was to fall back on HTTP when a client claimed it was unable to use HTTPS. When the server did so, no security would be applied, just as if the server wasn't running HTTPS at all. As ability to support HTTPS in browsers and client machines has become more common, there has been a push towards servers insisting on HTTPS, and refusing to talk to clients who can't or won't speak HTTPS. This approach is called HSTS (HTTP Strict Transport Security). HSTS is an option for a web site. If the web site decides it will support HSTS, all interactions with it will be cryptographically secured for any client. Clients who can't or won't accept HTTPS will not be allowed to interact with such a web site. HSTS is used by a number of major web sites, including Google's `google.com` domain, but is far from ubiquitous as of 2020.

While HTTPS is primarily intended to help secure web browsing, it is sometimes used to secure other kinds of communications. Some developers have leveraged HTTP for purposes rather different than standard web browsing, and, for them, using HTTPS to secure their communications is both natural and cheap. However, you can only use HTTPS to secure your system if you commit to using HTTP as your application protocol, and HTTP was intended primarily to support a human-based activity. HTTP messages, for example, are typically encoded in ASCII and include substantial headers designed to support web browsing needs. You may be able to achieve far greater efficiency of your application by using SSL, rather than HTTPS. Or you can use SSH.

SSH

SSH stands for **Secure Shell** which accurately describes the original purpose of the program. SSH is available on Linux and other Unix systems, and to some extent on Windows systems. SSH was envisioned as a secure remote shell, but it has been developed into a more general tool for allowing secure interactions between computers. Most commonly this shell is used for command line interfaces, but SSH can support many other forms of secure remote interactions. For example, it can be used to protect remote X Windows sessions. Generally, TCP ports can be forwarded through SSH, providing a powerful method to protect interactions between remote systems.

SSH addresses many of the same problems seen by SSL, often in similar ways. Remote users must be authenticated, shared encryption keys must be established, integrity must be checked, and so on. SSH typically relies on public key cryptography and certificates to authenticate remote servers. Clients frequently do not have their own certificates and private keys, in which case providing a user ID and password is permitted. SSH supports other options for authentication not based on certificates or passwords, such as the use of authentication servers (such as Kerberos). Various ciphers (both for authentication and for symmetric encryption) are supported, and some form of negotiation is required between the client and the server to choose a suitable set.

A typical use of SSH provides a good example of a common general kind of network security vulnerability called a **man-in-the-middle** attack. This kind of attack occurs when two parties think they are communicating directly, but actually are communicating through a malicious third party without knowing it. That third party sees all of the messages passed between them, and can alter such messages or inject new messages without their knowledge[7].

Well-designed network security tools are immune to man-in-the-middle attacks of many types, but even a good tool like SSH can sometimes be subject to them. If you use SSH much, you might have encountered an example yourself. When you first use SSH to log into a remote machine you've never logged into before, you probably don't have the public key associated with that remote machine. How do you get it? Often, not through a certificate or any other secure means, but simply by asking the remote site to send it to you. Then you have its public key and away you go, securely authenticating that machine and setting up encrypted communications. But what if there's a man in the middle when you first attempt to log into the remote machine? In that case, when the remote machine sends you its public key, the man in the middle can discard the message containing the correct public key and substitute one containing his own public key. Now you think you have the public key for the remote server, but you actually have the public key of the man in the middle. That means the man in the middle can pose as the remote server and you'll never be the wiser. The folks who designed SSH were well aware of this problem, and if you ever do use SSH this way, up will pop a message warning you of the danger and asking if you want to go ahead despite the risk. Folk wisdom suggests that everyone always says "yes, go ahead" when they get this message, including network security professionals. For that matter, folk wisdom suggests that all messages warning a user of the possibility of insecure actions are always ignored, which should suggest to you just how much security benefit will arise from adding such confirmation messages to your system.

SSH is not built on SSL, but is a separate implementation. As a result, the two approaches each have their own bugs, features, and uses. A security flaw found in SSH will not necessarily have any impact on SSL, and vice versa.

57.8 Summary

Distributed systems are critical to modern computing, but are difficult to secure. The cornerstone of providing distributed system security tends to be ensuring that the insecure network connecting system components does not introduce new security problems. Messages sent between the components are encrypted and authenticated, protecting their privacy and integrity, and offering exclusive access to the distributed service to the intended users. Standard tools like SSL/TLS and public keys

[7]Think back to our aside on Diffie-Hellman key exchange and the fly in the ointment. That's a perfect case for a man-in-the-middle attack, since an attacker can perhaps exchange a key with one correct party, rather than the two correct parties exchanging a key, without being detected.

distributed through X.509 certificates are used to provide these security services. Passwords are often used to authenticate remote human users.

Symmetric cryptography is used for transport of most data, since it is cheaper than asymmetric cryptography. Often, symmetric keys are not shared by system participants before the communication starts, so the first step in the protocol is typically exchanging a symmetric key. As discussed in previous chapters, key secrecy is critical in proper use of cryptography, so care is required in the key distribution process. Diffie-Hellman key exchange is commonly used, but it still requires authentication to ensure that only the intended participants know the key.

As mentioned in earlier chapters, building your own cryptographic solutions is challenging and often leads to security failures. A variety of tools, including SSL/TLS, SSH, and HTTPS, have already tackled many of the challenging problems and made good progress in overcoming them. These tools can be used to build other systems, avoiding many of the pitfalls of building cryptography from scratch. However, proper use of even the best security tools depends on an understanding of the tool's purpose and limitations, so developing deeper knowledge of the way such tools can be integrated into one's system is vital to using them to their best advantage.

Remember that these tools only make limited security guarantees. They do not provide the same assurance that an operating system gets when it performs actions locally on hardware under its direct control. Thus, even when using good authentication and encryption tools properly, a system designer is well advised to think carefully about the implications of performing actions requested by a remote site, or providing sensitive information to that site. What happens beyond the boundary of the machine the OS controls is always uncertain and thus risky.

References

[H14] "The Heartbleed Bug" by `http://heartbleed.com/`. *A web page providing a wealth of detail on this particular vulnerability in the OpenSSL implementation of the SSL/TLS protocol.*

[I12] "Information technology – Open Systems Interconnection – The Directory: Public-key and Attribute Certificate Frameworks" ITU-T, 2012. *The ITU-T document describing the format and use of an X.509 certificate. Not recommended for light bedtime reading, but here's where it's all defined.*

[NT94] "Kerberos: An authentication service for computer networks" by B. Clifford Neuman and Theodore Ts'o. IEEE Communications Magazine, Volume 32, No. 9, 1994. *An early paper on Kerberos by its main developers. There have been new versions of the system and many enhancements and bug fixes, but this paper is still a good discussion of the intricacies of the system.*

[P16] "The International PGP Home Page" `http://www.pgpi.org`, 2016. *A page that links to lots of useful stuff related to PGP, including downloads of free versions of the software, documentation, and discussion of issues related to it.*