

## 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<sup>1</sup>.*

**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!*

---

<sup>1</sup> According to this website ([http://www.barrypopik.com/index.php/new\\_york\\_city/entry/tell\\_me\\_and\\_i\\_forget.teach\\_me\\_and\\_i\\_may\\_remember.involve\\_me\\_and\\_i\\_will\\_lear/](http://www.barrypopik.com/index.php/new_york_city/entry/tell_me_and_i_forget.teach_me_and_i_may_remember.involve_me_and_i_will_lear/)), Confucian philosopher Xunzi said "Not having heard something is not as good as having heard it; having heard it is not as good as having seen it; having seen it is not as good as knowing it; knowing it is not as good as putting it into practice." Later on, the wisdom got attached to Confucius for some reason. Thanks to Jiao Dong (Rutgers) for telling us!

## Introduction to Operating Systems

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

So what happens when a program runs?

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

Thus, we have just described the basics of the **Von Neumann** model of computing<sup>2</sup>. 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

---

<sup>1</sup>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.

<sup>2</sup>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)**<sup>3</sup>, 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.

---

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

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

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

## 2.1 Virtualizing the CPU

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

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

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

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

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

```

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

```

Figure 2.2: Running Many Programs At Once

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

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

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

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

---

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

```

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

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

## 2.2 Virtualizing Memory

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

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

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

```

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

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

```

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

```

Figure 2.4: Running The Memory Program Multiple Times

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

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

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

## 2.3 Concurrency

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

---

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

```

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

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

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

Although you might not understand this example fully at the moment (and we'll learn a lot more about it in later chapters, in the section of the book on concurrency), the basic idea is simple. The main program creates two **threads** using `Pthread.create()`<sup>6</sup>. 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`

---

<sup>6</sup>The actual call should be to lower-case `pthread.create()`; the upper-case version is our own wrapper that calls `pthread.create()` and makes sure that the return code indicates that the call succeeded. See the code for details.

**THE CRUX OF THE PROBLEM:****HOW TO BUILD CORRECT CONCURRENT PROGRAMS**

When there are many concurrently executing threads within the same memory space, how can we build a correctly working program? What primitives are needed from the OS? What mechanisms should be provided by the hardware? How can we use them to solve the problems of concurrency?

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

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

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

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

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

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

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

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

## 2.4 Persistence

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

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

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

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

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

---

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

### THE CRUX OF THE PROBLEM: HOW TO STORE DATA PERSISTENTLY

The file system is the part of the OS in charge of managing persistent data. What techniques are needed to do so correctly? What mechanisms and policies are required to do so with high performance? How is reliability achieved, in the face of failures in hardware and software?

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

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

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

---

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

## 2.5 Design Goals

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

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

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

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

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

---

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

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

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

## 2.6 Some History

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

### Early Operating Systems: Just Libraries

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

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

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

### Beyond Libraries: Protection

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

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

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

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

## The Era of Multiprogramming

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

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

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

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

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

## The Modern Era

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

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

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

---

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

### ASIDE: THE IMPORTANCE OF UNIX

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

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

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

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

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

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

### ASIDE: AND THEN CAME LINUX

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

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

## 2.7 Summary

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

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

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

## References

- [BS+09] "Tolerating File-System Mistakes with EnvyFS"  
 Lakshmi N. Bairavasundaram, Swaminathan Sundararaman, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau  
 USENIX '09, San Diego, CA, June 2009  
*A fun paper about using multiple file systems at once to tolerate a mistake in any one of them.*
- [BH00] "The Evolution of Operating Systems"  
 P. Brinch Hansen  
 In Classic Operating Systems: From Batch Processing to Distributed Systems  
 Springer-Verlag, New York, 2000  
*This essay provides an intro to a wonderful collection of papers about historically significant systems.*
- [B+72] "TENEX, A Paged Time Sharing System for the PDP-10"  
 Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, Raymond S. Tomlinson  
 CACM, Volume 15, Number 3, March 1972  
*TENEX has much of the machinery found in modern operating systems; read more about it to see how much innovation was already in place in the early 1970's.*
- [B75] "The Mythical Man-Month"  
 Fred Brooks  
 Addison-Wesley, 1975  
*A classic text on software engineering; well worth the read.*
- [BOH10] "Computer Systems: A Programmer's Perspective"  
 Randal E. Bryant and David R. O'Hallaron  
 Addison-Wesley, 2010  
*Another great intro to how computer systems work. Has a little bit of overlap with this book — so if you'd like, you can skip the last few chapters of that book, or simply read them to get a different perspective on some of the same material. After all, one good way to build up your own knowledge is to hear as many other perspectives as possible, and then develop your own opinion and thoughts on the matter. You know, by thinking!*
- [K+61] "One-Level Storage System"  
 T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner  
 IRE Transactions on Electronic Computers, April 1962  
*The Atlas pioneered much of what you see in modern systems. However, this paper is not the best read. If you were to only read one, you might try the historical perspective below [L78].*
- [L78] "The Manchester Mark I and Atlas: A Historical Perspective"  
 S. H. Lavinton  
 Communications of the ACM archive  
 Volume 21, Issue 1 (January 1978), pages 4-12  
*A nice piece of history on the early development of computer systems and the pioneering efforts of the Atlas. Of course, one could go back and read the Atlas papers themselves, but this paper provides a great overview and adds some historical perspective.*
- [O72] "The Multics System: An Examination of its Structure"  
 Elliott Organick, 1972  
*A great overview of Multics. So many good ideas, and yet it was an over-designed system, shooting for too much, and thus never really worked as expected. A classic example of what Fred Brooks would call the "second-system effect" [B75].*

[PP03] "Introduction to Computing Systems:  
From Bits and Gates to C and Beyond"

Yale N. Patt and Sanjay J. Patel

McGraw-Hill, 2003

*One of our favorite intro to computing systems books. Starts at transistors and gets you all the way up to C; the early material is particularly great.*

[RT74] "The UNIX Time-Sharing System"

Dennis M. Ritchie and Ken Thompson

CACM, Volume 17, Number 7, July 1974, pages 365-375

*A great summary of UNIX written as it was taking over the world of computing, by the people who wrote it.*

[S68] "SDS 940 Time-Sharing System"

Scientific Data Systems Inc.

TECHNICAL MANUAL, SDS 90 11168 August 1968

Available: <http://goo.gl/EN0Zrn>

*Yes, a technical manual was the best we could find. But it is fascinating to read these old system documents, and see how much was already in place in the late 1960's. One of the minds behind the Berkeley Time-Sharing System (which eventually became the SDS system) was Butler Lampson, who later won a Turing award for his contributions in systems.*

[SS+10] "Membrane: Operating System Support for Restartable File Systems"

Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale,

Andrea C. Arpacı-Dusseau, Remzi H. Arpacı-Dusseau, Michael M. Swift

FAST '10, San Jose, CA, February 2010

*The great thing about writing your own class notes: you can advertise your own research. But this paper is actually pretty neat — when a file system hits a bug and crashes, Membrane auto-magically restarts it, all without applications or the rest of the system being affected.*

# Systems

# 1

## CHAPTER CONTENTS

<b>Overview .....</b>	<b>2</b>
<b>1.1 Systems and Complexity .....</b>	<b>3</b>
1.1.1 Common Problems of Systems in Many Fields.....	3
1.1.2 Systems, Components, Interfaces, and Environments.....	8
1.1.3 Complexity .....	10
<b>1.2 Sources of Complexity .....</b>	<b>13</b>
1.2.1 Cascading and Interacting Requirements .....	13
1.2.2 Maintaining High Utilization.....	17
<b>1.3 Coping with Complexity I .....</b>	<b>19</b>
1.3.1 Modularity .....	19
1.3.2 Abstraction .....	20
1.3.3 Layering .....	24
1.3.4 Hierarchy .....	25
1.3.5 Putting it Back Together: Names make Connections.....	26
<b>1.4 Computer Systems are the Same but Different.....</b>	<b>27</b>
1.4.1 Computer Systems have no Nearby Bounds on Composition.....	28
1.4.2 $d(\text{technology})/dt$ is Unprecedented .....	31
<b>1.5 Coping With Complexity II .....</b>	<b>35</b>
1.5.1 Why Modularity, Abstraction, Layering, and Hierarchy aren't Enough .....	36
1.5.2 Iteration .....	36
1.5.3 Keep it Simple.....	39
<b>What The Rest of this Book is About.....</b>	<b>40</b>
<b>Exercises .....</b>	<b>41</b>

## OVERVIEW

This book is about computer systems, and this chapter introduces some of the vocabulary and concepts used in designing computer systems. It also introduces “systems perspective”, a way of thinking about systems that is global and encompassing rather than focused on particular issues. A full appreciation of this way of thinking can’t really be captured in a short summary, so this chapter is actually just a preview of ideas that will be developed in depth in succeeding chapters.

The usual course of study of computer science and engineering begins with linguistic constructs for describing computations (software) and physical constructs for realizing computations (hardware). It then branches, focusing, for example, on the theory of computation, artificial intelligence, or the design of systems, which itself is usually divided into specialties: operating systems, transaction and database systems, computer architecture, software engineering, compilers, computer networks, security, and reliability. Rather than immediately tackling one of those specialties, we assume that the reader has completed the introductory courses on software and hardware, and we begin a broad study of computer systems that supports the entire range of systems specialties.

Many interesting applications of computers require

- fault tolerance
- coordination of concurrent activities
- geographically separated but linked data
- vast quantities of stored information
- protection from mistakes and intentional attacks
- interactions with many people

To develop applications that have these requirements, the designer must look beyond the software and hardware and view the computer system as a whole. In doing so, the designer encounters many new problems—so many that the limit on the scope of computer systems generally arises neither from laws of physics nor from theoretical impossibility, but rather from limitations of human understanding.

Some of these same problems have counterparts, or at least analogs, in other systems that have, at most, only incidental involvement of computers. The study of systems is one place where computer engineering can take advantage of knowledge from other engineering areas: civil engineering (bridges and skyscrapers), urban planning (the

Much wisdom about systems that has accumulated over the centuries is passed along in the form of folklore, maxims, aphorisms and quotations. Some of that wisdom is captured in the boxes at the bottom of these pages.

Everything should be made as simple as possible, but no simpler.

— commonly attributed to Albert Einstein; it is actually a paraphrase of a comment he made in a 1933 lecture at Oxford.

design of cities), mechanical engineering (automobiles and air conditioning), aviation and space flight, electrical engineering, and even ecology and political science. We start by looking at some of those common problems. Then we will examine two ways in which computer systems pose problems that are quite different. Don't worry if some of the examples are of things you have never encountered or are only dimly aware of. The sole purpose of the examples is to illustrate the range of considerations and similarities across different kinds of systems.

As we proceed in this chapter and throughout the book, we shall point out a series of *system design principles*, which are rules of thumb that usually apply to a diverse range of situations. Design principles are not immutable laws, but rather guidelines that capture wisdom and experience and that can help a designer avoid making mistakes. The astute reader will quickly realize that sometimes a tension, even to the point of contradiction, exists between different design principles. Nevertheless, if a designer finds that he or she is violating a design principle, it is a good idea to review the situation carefully.

At the first encounter of a design principle, the text displays it prominently. Here is an example, found on [page 16](#).

---

### Avoid excessive generality

*If it's good for everything, it's good for nothing.*

---

Each design principle thus has a formal title (“Avoid excessive generality”) and a brief informal description (“If it's good for . . .”), which are intended to help recall the principle. Most design principles will show up several times, in different contexts, which is one reason why they are useful. The text highlights later encounters of a principle such as: *avoid excessive generality*. A list of all of the design principles in the book can be found on the inside front cover and also in the index, under “Design principles”.

The remaining sections of this chapter discuss common problems of systems, the sources of those problems, and techniques for coping with them.

---

## 1.1 SYSTEMS AND COMPLEXITY

### 1.1.1 Common Problems of Systems in Many Fields

The problems one encounters in these many kinds of systems can usefully be divided into four categories: *emergent properties*, *propagation of effects*, *incommensurate scaling*, and *trade-offs*.

Seek simplicity and distrust it.

— Alfred North Whitehead, *The Concept of Nature* (1920)

### 1.1.1.1 *Emergent Properties*

*Emergent properties* are properties that are not evident in the individual components of a system, but show up when combining those components, so they might also be called surprises. Emergent properties abound in most systems, although there can always be a (fruitless) argument about whether or not careful enough prior analysis of the components might have allowed prediction of the surprise. It is wise to avoid this argument and instead focus on an unalterable fact of life: some things turn up only when a system is built.

Some examples of emergent properties are well known. The behavior of a committee or a jury often surprises outside observers. The group develops a way of thinking that could not have been predicted from knowledge about the individuals. (The concept of—and the label for—emergent properties originated in sociology.) When the Millennium Bridge for pedestrians over the River Thames in London opened, its designers had to close it after only a few days. They were surprised to discover that pedestrians synchronize their footsteps when the bridge sways, causing it to sway even more. Interconnection of several electric power companies to allow load sharing helps reduce the frequency of power failures, but when a failure finally occurs it may take down the entire interconnected structure. The political surprise is that the number of customers affected may be large enough to attract the unwanted attention of government regulators.

### 1.1.1.2 *Propagation of Effects*

The electric power inter-tie also illustrates the second category of system problems—*propagation of effects*—when a tree falling on a power line in Oregon leads to the lights going out in New Mexico, 1000 miles away. What looks at first to be a small disruption or a local change can have effects that reach from one end of a system to the other. An important requirement in most system designs is to limit the impact of failures. As another example of propagation of effects, consider an automobile designer's decision to change the tire size on a production model car from 13 to 15 inches. The reason for making the change might have been to improve the ride. On further analysis, this change leads to many other changes: redesigning the wheel wells, enlarging the spare tire space, rearranging the trunk that holds the spare tire, and moving the back seat forward slightly to accommodate the trunk redesign. The seat change makes knee room in the back seat too small, so the backs of the seats must be made thinner, which in turn reduces the comfort that was the original reason for changing the tire size, and it may also reduce safety in a collision. The extra weight of the trunk and rear seat design means that stiffer rear springs are now needed. The rear axle ratio must be modified to keep the force delivered to the road by the wheels correct, and the speedometer gearing must be changed to agree with the new tire size and axle ratio.

Those effects are the obvious ones. In complicated systems, as the analysis continues, more distant and subtle effects normally appear. As a typical example, the

Our life is frittered away by detail . . . simplicity, simplicity, simplicity!

— Henry David Thoreau, *Walden; or, Life in the Woods* (1854)

automobile manufacturer may find that the statewide purchasing office for Texas does not currently have a certified supplier for replacement tires of the larger size. Thus there will probably be no sales of cars to the Texas government for two years, which is the length of time it takes to add a supplier onto the certified list. Folk wisdom characterizes propagation of effects as: "There are no small changes in a large system".

### 1.1.1.3 Incommensurate Scaling

The third characteristic problem encountered in the study of systems is *incommensurate scaling*: as a system increases in size or speed, not all parts of it follow the same scaling rules, so things stop working. The mathematical description of this problem is that different parts of the system exhibit different orders of growth. Some examples:

- Galileo observed that "nature cannot produce a ... giant ten times taller than an ordinary man unless by ... greatly altering the proportions of his limbs and especially of his bones, which would have to be considerably enlarged over the ordinary" [*Discourses and Mathematical Demonstrations on Two New Sciences*, second day, Leiden, 1638]. In a classic 1928 paper, "On being the right size" [see Suggestions for Further Reading 1.4.1], J. B. S. Haldane uses the example of a mouse, which, if scaled up to the size of an elephant, would collapse of its own weight. For both examples, the reason is that weight grows with volume, which is proportional to the cube of linear size, but bone strength, which depends primarily on cross-sectional area, grows only with the square of linear size. Thus a real elephant requires a skeletal arrangement that is quite different from that of a scaled-up mouse.
- The Egyptian architect Sneferu tried to build larger and larger pyramids. Unfortunately, the facing fell off the pyramid at Meidum, and the ceiling of the burial chamber of the pyramid at Dashur cracked. He later figured out that he could escalate a pyramid to the size of the pyramids at Giza by lowering the ratio of the pyramid's height to its width. The reason this solution worked has apparently never been completely analyzed, but it seems likely that incommensurate scaling was involved—the weight of a pyramid increases with the cube of its linear size, while the strength of the rock used to create the ceiling of a burial chamber increases only with the area of its cross-section, which grows with the square.
- The captain of a modern oil supertanker finds that the ship is so massive that when underway at full speed it takes 12 miles to bring it to a straight line stop—but 12 miles is beyond the horizon as viewed from the ship's bridge (see [Sidebar 1.1](#) for the details).
- The height of a skyscraper is limited by the area of lower floors that must be devoted to providing access to the floors above. The amount of access area

By undue profundity we perplex and enfeeble thought.

— Edgar Allan Poe, "The Murders in the Rue Morgue" (1841)

**Sidebar 1.1 Stopping a Supertanker** A little geometry reveals that the distance to the visual horizon is proportional to the square root of the height of the bridge. That height (presumably) grows with the first power of the supertanker's linear dimension. The energy required to stop or turn a supertanker is proportional to its mass, which grows with the third power of its linear dimensions. The time required to deliver the stopping or turning energy is less clear, but pushing on the rudder and reversing the propellers are the only tools available, and both of those have surface area that grows with the square of the linear dimension.

Here is the bottom line: if we double the tanker's linear dimensions, the momentum goes up by a factor of 8, and the ability to deliver stopping or turning energy goes up by only a factor of 4, so we need to see twice as far ahead. Unfortunately, the horizon will be only 1.414 times as far away. Inevitably, there is some size for which visual navigation must fail.

required (for example, for elevators and stairs) is proportional to the number of people who have offices on higher floors. That number is in turn proportional to the number of higher floors multiplied by the usable area of each floor. If all floors have the same area, and the number of floors increases, at some point the bottom floor will be completely used up providing access to higher floors, so the bottom floor provides no added value (apart from being able to brag about the building's height). In practice, the economics of office real estate dictate that no more than 25% of the lowest floor be devoted to access.

Incommensurate scaling shows up in most systems. It is usually the factor that limits the size or speed range that a single system design can handle. On the other hand, one must be cautious with scaling arguments. They were used at the beginning of the twentieth century to support the claim that it was a waste of time to build airplanes (see [Sidebar 1.2](#)).

#### 1.1.1.4 Trade-offs

The fourth problem of system design is that many constraints present themselves as *trade-offs*. The general model of a trade-off begins with the observation that there is a limited amount of some form of goodness in the universe, and the design challenge is first to maximize that goodness, second to avoid wasting it, and third to allocate it to the places where it will help the most. One common form of trade-off is sometimes called the *waterbed effect*: pushing down on a problem at one point causes another problem to pop up somewhere else. For example, one can typically push a hardware circuit to run at a higher clock rate, but that change increases both power consumption and the risk of timing errors. It may be possible to reduce the risk of timing errors by making the circuit physically smaller, but then less area will be available to dissipate the heat caused

KISS: Keep It Simple, Stupid.

— traditional management folklore; source lost in the mists of time

**Sidebar 1.2 Why Airplanes can't Fly** The weight of an airplane grows with the third power of its linear dimension, but the lift, which is proportional to surface area, can grow only with the second power. Even if a small plane can be built, a larger one will never get off the ground.

This line of reasoning was used around 1900 by both physicists and engineers to argue that it was a waste of time to build heavier-than-air machines. Alexander Graham Bell proved that this argument wasn't the whole story by flying box kites in Maine in the summer of 1902. In his experiments he attached two box kites side by side, a configuration that doubled the lifting surface area, but also allowed removal of the redundant material and supports where the two kites touched. Thus, the lift-to-weight ratio actually improved as the scale increased. Bell published his results in "The tetrahedral principle in kite structure" [see Suggestions for Further Reading 1.4.2].

by the increased power consumption. Another common form of trade-off appears in *binary classification*, which arises, for example, in the design of smoke detectors, spam (unwanted commercial e-mail message) filters, database queries, and authentication devices. The general model of binary classification is that we wish to classify a set of things into two categories based on the presence or absence of some property, but we lack a direct measure of that property. We therefore instead identify and use some indirect measure, known as a *proxy*. Occasionally, this scheme misclassifies something. By adjusting parameters of the proxy, the designer may be able to reduce one class of mistakes (in the case of a smoke detector, unnoticed fires; for a spam filter, legitimate messages marked as spam), but only at the cost of increasing some other class of mistakes (for the smoke detector, false alarms; for the spam filter, spam marked as legitimate messages). Appendix A explores the binary classification trade-off in more detail. Much of a system designer's intellectual effort goes into evaluating various kinds of trade-offs.

Emergent properties, propagation of effects, incommensurate scaling, and trade-offs are issues that the designer must deal with in every system. The question is how to build useful computer systems in the face of such problems. Ideally, we would like to describe a constructive theory, one that allows the designer systematically to synthesize a system from its specifications and to make necessary trade-offs with precision, just as there are constructive theories in such fields as communications systems, linear control systems, and (to a certain extent) the design of bridges and skyscrapers. Unfortunately, in the case of computer systems, we find that we were apparently born too soon. Although our early arrival on the scene offers the challenge to develop the missing theory, the problem is quickly apparent—we work almost entirely by analyzing *ad hoc* examples rather than by synthesizing.

Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.

— Alan J. Perlis, "Epigrams in Programming" (1982)

So, in place of a well-organized theory, we use case studies. For each subtopic in this book, we shall begin by identifying requirements with the apparent intent of deriving the system structure from the requirements. Then, almost immediately we switch to case studies and work backwards to see how real, in-the-field systems meet the requirements that we have set. Along the way we point out where systematic approaches to synthesizing a system from its requirements are beginning to emerge, and we introduce representations, abstractions, and design principles that have proven useful in describing and building systems. The intended result of this study is insight into how designers create real systems.

### 1.1.2 Systems, Components, Interfaces, and Environments

*Webster's Third New International Dictionary, Unabridged*, defines a system as “a complex unity formed of many often diverse parts subject to a common plan or serving a common purpose.” Although this definition will do for casual use of the word, engineers usually prefer something a bit more concrete. We identify the “many often diverse parts” by naming them *components*. We identify the “unity” and “common plan” with the *interconnections* of the components, and we perceive the “common purpose” of a system to be to exhibit a certain behavior across its *interface* to an *environment*. Thus we formulate our technical definition: **A system is a set of interconnected components that has an expected behavior observed at the interface with its environment.**

The underlying idea of the concept of system is to divide all the things in the world into two groups: those under discussion and those not under discussion. Those things under discussion are part of the system—those that are not are part of the *environment*. For example, we might define the solar system as consisting of the sun, planets, asteroids, and comets. The environment of the solar system is the rest of the universe. (Indeed, the word “universe” is a synonym for environment.)

There are always interactions between a system and its environment; these interactions are the *interface* between the system and the environment. The interface between the solar system and the rest of the universe includes gravitational attraction for the nearest stars and the exchange of electromagnetic radiation. The primary interfaces of a personal computer typically include things such as a display, keyboard, speaker, network connection, and power cord, but there are also less obvious interfaces such as the atmospheric pressure, ambient temperature and humidity, and the electromagnetic noise environment.

One studies a system to predict its overall behavior, based on information about its components, their interconnections, and their individual behaviors. Identifying the components, however, depends on one’s point of view, which has two aspects, *purpose* and *granularity*. One may, with different purposes in mind, look at a system quite

And simplicity is the unavoidable price we must pay for reliability.

— Charles Anthony Richard Hoare, “Data Reliability” (1975)

**Sidebar 1.3 Terminology: Words used to Describe System Composition** Since systems can contain component subsystems that are themselves systems from a different point of view, decomposition of systems is recursive. To avoid recursion in their writing, authors and designers have come up with a long list of synonyms, all trying to capture this same concept: *systems, subsystems, components, elements, constituents, objects, modules, submodules, assemblies, subassemblies*, and so on.

differently. One may also choose any of several different granularities. These choices affect one's identification of the components of the system in important ways.

To see how point of view can depend on purpose, consider two points of view of a jet aircraft as a system. The first looks at the aircraft as a flying object, in which the components of the system include the body, wings, control surfaces, and engines. The environment is the atmosphere and the earth, with interfaces consisting of gravity, engine thrust, and air drag. A second point of view looks at the aircraft as a passenger-handling system. Now, the components include seats, flight attendants, the air conditioning system, and the galley. The environment is the set of passengers, and the interfaces are the softness of the seats, the meals, and the air flowing from the air conditioning system.

In the first point of view, the aircraft as a flying object, the seats, flight attendants, and galley were present, but the designer considers them primarily as contributors of weight. Conversely, in the second point of view, as a passenger-handling system, the designer considers the engine as a source of noise and perhaps also exhaust fumes, and probably ignores the control surfaces on the wings. Thus, depending on point of view, we may choose to ignore or consolidate certain system components or interfaces.

The ability to choose granularity means that a component in one context may be an entire system in another. From an aircraft designer's point of view, a jet engine is a component that contributes weight, thrust, and perhaps drag. On the other hand, the manufacturer of the engine views it as a system in its own right, with many components—turbines, hydraulic pumps, bearings, afterburners, all of which interact in diverse ways to produce thrust—one interface with the environment of the engine. The airplane wing that supports the engine is a component of the aircraft system, but it is part of the environment of the engine system.

When a system in one context is a component in another, it is usually called a *subsystem* (but see Sidebar 1.3). The composition of systems from subsystems or decomposition of systems into subsystems can be carried on to as many levels as is useful.

In summary, then, to analyze a system one must establish a point of view to determine which things to consider as components, what the granularity of those

Pluralitas non est ponenda sine neccesitate. (Plurality should not be assumed without necessity.)

— William of Ockham (14th century. Popularly known as "Occam's razor," though the idea itself is said to appear in writings of greater antiquity.)

components should be, where the boundary of the system lies, and which interfaces between the system and its environment are of interest.

As we use the term, a *computer system* or an *information system* is a system intended to store, process, or communicate information under automatic control. Further, we are interested in systems that are predominantly digital. Here are some examples:

- a personal computer
- the onboard engine controller of an automobile
- the telephone system
- the Internet
- an airline ticket reservation system
- the space shuttle ground control system
- a World Wide Web site

At the same time we will sometimes find it useful to look at examples of nondigital and nonautomated information handling systems, such as the post office or library, for ideas and guidance.

### 1.1.3 Complexity

Webster's definition of "system" used the word "complex". Looking up that term, we find that *complex* means "difficult to understand". Lack of systematic understanding is the underlying feature of complexity. It follows that complexity is both a subjective and a relative concept. That is, one can argue that one system is more complex than another, but even though one can count up various things that seem to contribute to complexity, there is no unified measure. Even the argument that one system is more complex than another can be difficult to make compelling—again because of the lack of a unified measure. In place of such a measure, we can borrow a technique from medicine: describe a set of *signs* of complexity that can help confirm a diagnosis. As a corollary, we abandon hope of producing a definitive description of complexity. We must instead look for its signs, and if enough appear, argue that complexity is present. To that end, here are five signs of complexity:

- 1. Large number of components.** Sheer size certainly affects our view of whether or not a system rates the description "complex".
- 2. Large number of interconnections.** Even a few components may be interconnected in an unmanageably large number of ways. For example, the Sun and the known planets comprise only a few components, but every one has gravitational

Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher. (It is as if perfection be attained not when there is nothing more to add, but when there is nothing more to take away.)

— Antoine de Saint-Exupéry, *Terre des Hommes* (1939)

attraction for every other, which leads to a set of equations that are unsolvable (in closed form) with present mathematical techniques. Worse, a small disturbance can, after a while, lead to dramatically different orbits. Because of this sensitivity to disturbance, the solar system is technically *chaotic*. Although there is no formal definition of chaos for computer systems, that term is often informally applied.

3. **Many irregularities.** By themselves, a large number of components and interconnections may still represent a simple system, if the components are repetitive and the interconnections are regular. However, a lack of regularity, as shown by the number of exceptions or by non-repetitive interconnection arrangements, strongly suggests complexity. Put another way, exceptions complicate understanding.
4. **A long description.** Looking at the best available description of the system one finds that it consists of a long laundry list of properties rather than a short, systematic specification that explains every aspect. Theoreticians formalize this idea by measuring what they call the “Kolmogorov complexity” of a computational object as the length of its shortest specification. To a certain extent, this sign may be merely a reflection of the previous three, although it emphasizes an important aspect of complexity: it is relative to understanding. On the other hand, lack of a methodical description may also indicate that the system is constructed of ill-fitting components, is poorly organized, or may have unpredictable behavior, any of which add complexity to both design and use.
5. **A team of designers, implementers, or maintainers.** Several people are required to understand, construct, or maintain the system. A fundamental issue in any system is whether or not it is simple enough for a single person to understand all of it. If not, it is a complex system because its description, construction, or maintenance will require not just technical expertise but also coordination and communication across a team.

Again, an example can illustrate: contrast a small-town library with a large university library. There is obviously a difference in scale: the university has more books, so the first sign is present. The second sign is more subtle: where the small library may have a catalog to guide the user, the university library may have not only a catalog,

'Tis the gift to be simple, 'tis the gift to be free,  
'Tis the gift to come down where we ought to be;  
And when we find ourselves in the place just right,  
'Twill be in the valley of love and delight.  
When true simplicity is gained  
To bow and to bend we shan't be ashamed;  
To turn, turn will be our delight,  
Till by turning, turning we come round right.

— *Simple Gifts*, traditional Shaker hymn

but also finding aids, readers' guides, abstracting services, journal indexes, and so on. Although these elaborations make the large library more useful (at least to the experienced user), they also complicate the task of adding a new item to the library: someone must add many interconnections (in this case, cross-references) so that the new item can be found in all the intended ways. The third sign, a large number of exceptions, is also apparent. Where the small library has only a few classifications (fiction, biography, nonfiction, and magazines) and a few exceptions (oversized books are kept over the newspaper rack), the university library is plagued with exceptions. Some books are oversized, others come on microfilm or on digital media, some books are rare or valuable and must be protected, the books that explain how to build a hydrogen bomb can be loaned only to certain patrons, some defy cataloging in any standard classification system. As for the fourth sign, any user of a large university library will confirm that there are no methodical rules for locating a piece of information and that library usage is an art, not a science.

Finally, the fifth sign of complexity, a staff of more than one person, is evident in the university library. Where many small towns do in fact have just one librarian—typically an energetic person who knows each book because at one time or another he or she has had occasion to touch it—the university library has not only many personnel, but even specialists who are familiar with only one facet of library operations, such as the microform collection.

The university library exhibits all five signs of complexity, but unanimity is not essential. On the other hand, the presence of only one or two of the signs may not make a compelling case for complexity. Systems considered in thermodynamics contain an unthinkably large number of components (elementary particles) and interactions, yet from the right point of view they do not qualify as complex because there is a simple, methodical description of their behavior. It is exactly when we lack such a simple, methodical description that we have complexity.

One objection to conceiving complexity as being based on the five signs is that all systems are indefinitely, perhaps infinitely, complex because the deeper one digs the more signs of complexity turn up. Thus, even the simplest digital computer is made of gates, which are made with transistors, which are made of silicon, which is composed of protons, neutrons, and electrons, which are composed of quarks, which some physicists suggest are describable as vibrating strings, and so on. We shall address this objection in a moment by limiting the depth of digging, a technique known as *abstraction*. The complexity that we are interested in and worried about is the complexity that remains despite the use of abstraction.

Whatever man builds . . . all of man's . . . efforts . . . invariably culminate in . . . a thing whose sole and guiding principle is . . . simplicity . . . perfection of invention touches hands with absence of invention, as if . . . [there] were a line that had not been invented but . . . [was] in the beginning . . . hidden by nature and in the end . . . found by the engineer.

— Antoine de Saint-Exupéry, *Terre des Hommes* (1939)

---

## 1.2 SOURCES OF COMPLEXITY

There are many sources of complexity, but two merit special mention. The first is in the number of requirements that the designer expects a system to meet. The second is one particular requirement: maintaining high utilization.

### 1.2.1 Cascading and Interacting Requirements

A primary source of complexity is just the list of requirements for a system. Each requirement, viewed by itself, may seem straightforward. Any particular requirement may even appear to add only easily tolerable complexity to an existing list of requirements. The problem is that the accumulation of many requirements adds not only their individual complexities but also complexities from their interactions. This interaction complexity arises from pressure for generality and exceptions that add complications, and it is made worse by change in individual requirements over time.

Most users of a personal computer have by now encountered some version of the following scenario: The vendor announces a new release of the program you use to manage your checkbook, and the new release has some feature that seems important or useful (e.g., it handles the latest on-line banking systems), so you order the program. Upon trying to install it, you discover that this new release requires a newer version of some shared library package. You track down that newer version and install it, only to find that the library package requires a newer version of the operating system, which you had not previously had any reason to install. Biting the bullet, you install the latest release of the operating system, and now the checkbook program works, but your add-on hard disk begins to act flaky. On investigation it turns out that the disk vendor's proprietary software is incompatible with the new operating system release. Unfortunately, the disk vendor is still debugging an update for the disk software, and the best thing available is a beta test version that will expire at the end of the month.

The underlying cause of this scenario is that the personal computer has been designed to meet many requirements: a well-organized file system, expandability of storage, ability to attach a variety of I/O devices, connection to a network, protection from malevolent persons elsewhere in the network, usability, reliability, low cost—the list goes on and on. Each of these requirements adds complexity of its own, and the interactions among them add still more complexity.

Similarly, the telephone system has, over the years, acquired a large number of line customizing features—call waiting, call return, call forwarding, originating and terminating call blocking, reverse billing, caller ID, caller ID blocking, anonymous call

When in doubt, make it stout, and of things you know about.

When in doubt, leave it out.

— folklore sayings from the automobile industry

**Sidebar 1.4 The Cast of Characters and Organizations** In concrete examples throughout this book, the reader will encounter a standard cast of characters named Alice, Bob, Charles, Dawn, Ella, and Felipe. Alice is usually the sender of a message, and Bob is its recipient. Charles is sometimes a mutual acquaintance of Alice and Bob. The others play various supporting roles, depending on the example. When we come to security, an adversarial character named Lucifer will appear. Lucifer's role is to crack the security measures and perhaps interfere with the presumably useful work of the other characters.

The book also introduces a few fictional organizations. There are two universities: Pedantic University, on the Internet at [Pedantic.edu](#), and The Institute of Scholarly Studies, at [Scholarly.edu](#). There are also four mythical commercial organizations on the Internet at [TrustUs.com](#), [ShopWithUs.com](#), [Awesome.net](#), and [Awful.net](#).

M.I.T. Professor Ronald Rivest introduced Alice and Bob to the literature of computer science in Suggestions for Further Reading 11.5.1. Any other resemblance to persons living or dead or organizations real or imaginary is purely coincidental.

rejection, do not disturb, vacation protection—again, the list goes on and on. These features interact in so many ways that there is a whole field of study of “feature interaction” in telephone systems. The study begins with debates over what *should* happen. For example, so-called 900 numbers have the feature called reverse billing—the called party can place a charge on the caller’s bill. Alice (Alice is the first character we have encountered in our cast of characters, described in Sidebar 1.4) has a feature that blocks outgoing calls to reverse billing numbers. Alice calls Bob, whose phone is forwarded to a 900 number. Should the call go through, and if so, which party should pay for it, Bob or Alice? There are three interacting features, and at least four different possibilities: block the call, allow the call and charge it to Bob, ring Bob’s phone, or add yet another feature that (for a monthly fee) lets Bob choose the outcome.

The examples suggest that there is an underlying principle at work. We call it the:

---

#### Principle of escalating complexity

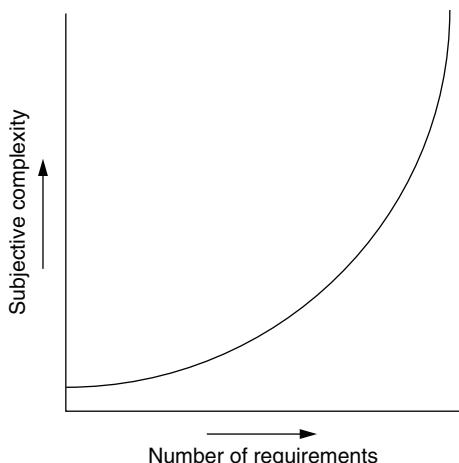
*Adding a requirement increases complexity out of proportion.*

---

The principle is subjective because complexity itself is subjective—its magnitude is in the mind of the beholder. Figure 1.1 provides a graphical interpretation of the

Perfection must be reached by degrees; she requires the slow hand of time.

— attributed to François-Marie Arouet (Voltaire)

**FIGURE 1.1**

The principle of escalating complexity.

particular supplier. That component may be less durable, heavier, or not as available as one from another supplier. Those properties may not prevent its use, but they add complexity to other parts of the system that have to be designed to compensate.

Meeting many requirements with a single design is sometimes expressed as a need for *generality*. Generality may be loosely defined as “applying to a variety of circumstances.” Unfortunately, generality contributes to complexity, so it comes with a trade-off, and the designer must use good judgment to decide how much of the generality is actually *wanted*. As an extreme example, an automobile with four independent steering wheels, each controlling one tire, offers some kind of ultimate in generality, almost all of which is unwanted. Here, both the aspect of unwantedness and the resulting complexity of guidance of the auto are obvious enough, but in many cases both of these aspects are more difficult to assess: How much does a proposed form of generality complicate the system, and to what extent is that generality really useful? Unwanted generality also contributes to complexity indirectly: users of a system with excessive generality will adopt styles of usage that simplify and suppress generality that they do not need. Different users may adopt different styles and then discover that they cannot easily exchange ideas with one another. Anyone who tries to use a personal computer customized by someone else will notice this problem.

Periodically, someone tries to design a vehicle that one can drive on the highway, fly, and use as a boat, but the result of such a general design does not seem to work

principle. Perhaps the most important thing to recognize in studying this figure is that the complexity barrier is soft: as you add features and requirements, you don’t hit a solid roadblock to warn you to stop adding. It just gets worse.

As the number of requirements grows, so can the number of exceptions and thus the complications. It is the incredible number of special cases in the United States tax code that makes filling out an income tax return a complex job. The impact of any one exception may be minor, but the cumulative impact of many interacting exceptions can make a system so complex that no one can understand it. Complications also can arise from outside requirements such as insistence that a certain component must come from a

The best is the enemy of the good.

— François-Marie Arouet (Voltaire), *Dictionnaire Philosophique* (1764)

well in any of the intended modes of transport. To help counter excessive generality, experience suggests another design principle.\*

---

### Avoid excessive generality

*If it is good for everything, it is good for nothing.*

---

There is a tension between exceptions and generality. Part of the art of designing a subsystem is to make its features general enough to minimize the number of exceptions that must be handled as special cases. This area is one where the judgment of the system designer is most evident.

Counteracting the effects of incommensurate scaling can be an additional source of complexity. Haldane, in his essay “On being the right size”, points out that small organisms such as insects absorb enough oxygen to survive through their skins, but larger organisms, which require an amount of oxygen proportional to the cube of their linear size, don’t have enough surface area. To compensate for this incommensurate scaling, they add complexity in the form of lungs and blood vessels to absorb and deliver oxygen throughout their bodies. In the case of computers, the programmer of a 4-bit microprocessor to control a toaster can in a few days successfully write the needed code entirely with binary numbers, while the programmer of a video game with a 64-bit processor and 40 gigabytes of supporting data requires an extensive array of tools—compilers, image or video editors, special effects generators, and the like, as well as an operating system, to be able to get the job done within a lifetime. Incommensurate scaling has required employment of a far more complex set of tools.

Finally, a major source of complexity is that requirements *change*. System designs that are successful usually remain in use for a long time, during which the environment of the system changes. Improvements in hardware technology may lead the system maintainers to want to upgrade to faster, cheaper, or more reliable equipment. Meanwhile, knowledge of how to maintain the older equipment (and the supply of spare parts) may be disappearing. As users accumulate experience with the system, it becomes clearer that some additional requirements should have been part of the design and that some of the original requirements were less important than originally thought. Often a system will expand in scale, sometimes far beyond the vision of its original designers.

In each of these cases, the ground rules and assumptions that the original designers used to develop the system begin to lose their relevance. The system designers

---

\*Computer industry consultant (and erstwhile instructor of the course for which this textbook was written) Michael Hammer suggested the informal version of this design principle.

A complex system that works is invariably found to have evolved from a simple system that works.

— John Gall, *Systemantics* (1975)

may have foreseen some environmental changes, but there were other changes they probably did not anticipate. As changes to meet unforeseen requirements occur, they usually add complexity. Because it can be difficult to change the architecture of a deployed system (Section 1.3 explains why), there is a powerful incentive to make changes within the existing architecture, whether or not that is the best thing to do. Propagation of effects can amplify the problems caused by change because more distant effects of a change may not be noticed until someone invokes some rarely used feature. When those distant effects finally do surface, the maintainer may again find it easiest to deal with them locally, perhaps by adding exceptions. Incommensurate scaling effects begin to dominate behavior when a later maintainer scales a system up in size or replaces the underpinnings with faster hardware. Again, the first response to these effects is usually to make local changes (sometimes called *patches*) to counteract them rather than to make fundamental changes in design that would require changing several modules or changing interfaces between modules.

A closely related problem is that as systems grow in complexity with the passage of time, even the simplest change, such as to repair a bug, has an increasing risk of introducing another bug because complexity tends to obscure the full impact of the repair. A common phenomenon in older systems is that the number of bugs introduced by a bug fix release may exceed the number of bugs fixed by that release.\*

The bottom line is that as systems age, they tend to accumulate changes that make them more complex. The lifetime of a system is usually limited by the complexity that accumulates as it evolves farther and farther from its original design.

## 1.2.2 Maintaining High Utilization

One requirement by itself is frequently a specific source of complexity. It starts with a desire for high performance or high efficiency. Whenever a scarce resource is involved, an effort arises to keep its *utilization* high.

Consider, for example, a single-track railroad line running through a long, narrow canyon.<sup>†</sup> To improve the utilization of the single track, and push more traffic through, one might allow trains to run both ways at the same time by installing a switch and a short side track in a wide spot about halfway through the canyon. Then, if one is careful in scheduling, trains going in opposite directions will meet at the side track, where

---

\*This phenomenon was documented by Laszlo A. Belady and Meir M. Lehman in “A model of large program development”, *IBM Systems Journal* 15, 3 (1976), pages 225–252.

<sup>†</sup>Michael D. Schroeder suggested this example of a railroad line in a canyon.

Een schip op’t droogh gezeylt, dat is een seeker baken. (A ship, sailed on to dry land, that is a certain beacon. Learn from the mistakes of others.)

— Jacob Cats, *Mirror on Old and New Times* (1632), based on a Dutch proverb

they can pass each other, effectively doubling the number of trains that the track can carry each day. However, the train operations are now much more complex than they used to be. If either train is delayed, the schedules of both are disrupted. A signaling system needs to be installed because human schedulers or operators may make mistakes. And—an emergent property—the trains now have a limit on their length. If two trains are to pass in the middle, at least one of them must be short enough to pull completely onto the side track.

The train in the canyon is a good illustration of how efforts to increase utilization can increase complexity. When striving for higher utilization, one usually encounters a general design principle that economists call

---

#### The law of diminishing returns

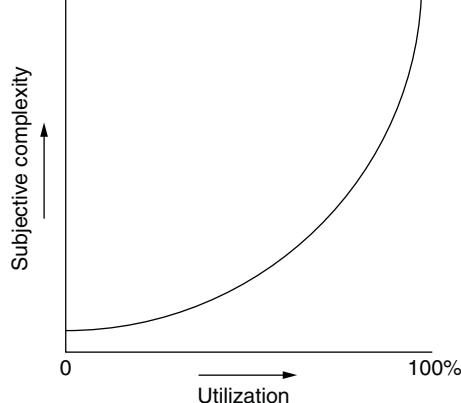
*The more one improves some measure of goodness, the more effort the next improvement will require.*

---

This phenomenon is particularly noticeable in attempts to use resources more efficiently: the more completely one tries to use a scarce resource, the greater the complexity of the strategies for use, allocation, and distribution. Thus a rarely used street

intersection requires no traffic control beyond a rule that the car on the right has the right-of-way. As usage increases, one must apply progressively more complex measures: stop signs, then traffic lights, then marked turning lanes with multiphase lights, then vehicle sensors to control the lights. As traffic in and out of an airport nears the airport's capacity, measures such as stacking planes, holding them on the ground at distant airports, or coordinated scheduling among several airlines must be taken. As a general rule, the more one tries to increase utilization of a limited resource, the greater the complexity (see Figure 1.2).

The perceptive reader will notice that Figures 1.1 and 1.2 are identical. It would



**FIGURE 1.2**

An example of diminishing returns: complexity grows with increasing utilization.

It is impossible to foresee the consequences of being clever.

— Christopher Strachey, as reported by Roger Needham

be useful to memorize this figure because some version of it can be used to describe many different things about systems.

## 1.3 COPING WITH COMPLEXITY I

As one might expect, with many fields contributing examples of systems with common problems and sources of complexity, some common techniques for coping with complexity have emerged. These techniques can be loosely divided into four general categories: *modularity*, *abstraction*, *layering*, and *hierarchy*. The following sections sketch the general method of each of the techniques. In later chapters many examples of each technique will emerge. It is only by studying those examples that their value will become clear.

### 1.3.1 Modularity

The simplest, most important tool for reducing complexity is the divide-and-conquer technique: analyze or design the system as a collection of interacting subsystems, called *modules*. The power of this technique lies primarily in being able to consider interactions among the components within a module without simultaneously thinking about the components that are inside other modules.

To see the impact of reducing interactions, consider the debugging of a large program with, say,  $N$  statements. Assume that the number of bugs in the program is proportional to its size and the bugs are randomly distributed throughout the code. The programmer compiles the program, runs it, notices a bug, finds and fixes the bug, and recompiles before looking for the next bug. Assume also that the time it takes to find a bug in a program is roughly proportional to the size of the program. We can then model the time spent debugging:

$$\begin{aligned} \text{BugCount} &\sim N \\ \text{DebugTime} &\sim N \times \text{BugCount} \\ &\sim N^2 \end{aligned}$$

Unfortunately, the debugging time grows proportional to the square of the program size.

Now suppose that the programmer divides the program into  $K$  modules, each of roughly equal size, so that each module contains  $N/K$  statements. To the extent that the modules implement independent features, one hopes that discovery of a bug usually will require examining only one module. The time required to debug any one module is thus reduced in two ways: the smaller module can be debugged faster, and

Plan to throw one away; you will, anyhow.

— Frederick P. Brooks, *The Mythical Man Month* (1974)

since there are fewer bugs in smaller programs, any one module will not need to be debugged as many times. These two effects are partially offset by the need to debug all  $K$  modules. Thus our model of the time required to debug the system of  $K$  modules becomes

$$\begin{aligned} \text{DebugTime} &\sim \left(\frac{N}{K}\right)^2 \times K \\ &\sim \frac{N^2}{K} \end{aligned}$$

Modularization into  $K$  components thus reduces debugging time by a factor of  $K$ . Although the detailed mechanism by which modularity reduces effort differs from system to system, this property of modularity is universal. For this reason, one finds modularity in every large system.

The feature of modularity that we are taking advantage of here is that it is easy to replace an inferior module with an improved one, thus allowing incremental improvement of a system without completely rebuilding it. Modularity thus helps control the complexity caused by change. This feature applies not only to debugging but to all aspects of system improvement and evolution. At the same time, it is important to recognize a design principle associated with modularity, which we may call

#### **The unyielding foundations rule**

*It is easier to change a module than to change the modularity.*

The reason is that once an interface has been used by another module, changing the interface requires replacing at least two modules. If an interface is used by many modules, changing it requires replacing all of those modules simultaneously. For this reason, it is particularly important to get the modularity right.

Whole books have been written about modularity and the good things it brings. Sidebar 1.5 describes one of those books.

### **1.3.2 Abstraction**

An important assumption in the numerical example of the effect of modularity on debugging time may not hold up in practice: that discovery of a bug should usually lead to examining just one module. For that assumption to hold true, there is a further requirement: there must be little or no propagation of effects from one module to

The purpose of computing is insight, not numbers.

— Richard W. Hamming, *Numerical Methods for Scientists and Engineers* (1962)

**Sidebar 1.5 How Modularity Reshaped the Computer Industry** Two Harvard Business School professors, Carliss Baldwin and Kim Clark, have written a whole book about modularity.\* It discusses many things, but one of the most interesting is its explanation of a major transition in the computer business. In the 1960s, computer systems were a vertically integrated industry. That is, IBM, Burroughs, Honeywell, and several others each provided top-to-bottom systems and support, offering processors, memory, storage, operating systems, applications, sales, and maintenance; IBM even manufactured its own chips. By the 1990s, the industry had transformed into a horizontally organized one in which Intel sells processors, Micron sells memory, Seagate sells disks, Microsoft sells operating systems, Adobe sells text and image applications, Oracle sells database systems, and Gateway and Dell assemble boxes called “computers” out of components provided by the other players.

Carliss Baldwin and Kim Clark explain this transition as an example of modularity in action. The companies that created vertically integrated product lines immediately found complexity running amok, and they concluded that the only effective way to control it was to modularize their products. After a few experiments with wrong modularities (IBM originally designed different computers for business and for scientific applications), they eventually hit on effective ways of splitting things up and thereby keeping their development costs and delivery schedules under control:

- IBM developed the System/360 architecture specification, which could apply to machines of widely ranging performance. This modularity allowed any software to run on any size processor. IBM also developed a standard I/O bus and disk interface, so that any I/O device or disk manufactured by IBM could be attached to any IBM computer.
- Digital Equipment Corporation developed the PDP-11 family, which, with improving technology, could simultaneously be driven down in price toward the PDP-11/03 and up in function toward the PDP-11/70. A hardware-assisted emulation strategy for missing hardware instructions on the smaller machines allowed applications written for any machine to run on any other machine in the family. Digital also developed an I/O architecture, the UNIBUS®, that allowed any I/O device to attach to any PDP-11 model.

The long-range result was that once this modularity was defined and proven to be effective, other vendors were able to jump in and turn each module into a distinct business. The result is the computer industry since the 1990s, which is remarkably horizontal, especially considering its rather different shape only 20 years earlier.

(Sidebar continues)

---

\*Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity* [see Suggestions for Further Reading 1.3.7]. Warning: the authors use the word “modularity” to mean all of modularity, abstraction, layering, and hierarchy.

Carliss Baldwin and Kim Clark also observe, more generally, that a market economy is characterized by modularity. Rather than having a self-supporting farm family that does everything for itself, a market economy has coopers, tinkers, blacksmiths, stables, dressmakers, and so on, each being more productive in a modular specialty; all selling things to one another using a universal interface—money.

another. Although there are lots of ways of dividing a system up into modules, some of these ways will prove to be better than others—“according to the natural formation, where the joint is, not breaking any part as a bad carver might” (Plato, *Phaedrus* 265e, Benjamin Jowett translation).

Thus the best divisions usually follow natural or effective boundaries. They are characterized by fewer interactions among modules and by less propagation of effects from one module to another. More generally, they are characterized by the ability of any module to treat all the others entirely on the basis of their external specifications, without need for knowledge about what goes on inside. This additional requirement on modularity is called *abstraction*. Abstraction is separation of interface from internals, of specification from implementation. Because abstraction nearly always accompanies modularity, some authors do not make any distinction between the two ideas. One sometimes sees the term *functional modularity* used to mean modularity with abstraction.

Thus one purchases a DVD player planning to view it as a device with a dozen or so buttons on the front panel and hoping never to look inside. If one had to know the details of the internal design of a television set in order to choose a compatible DVD player, no one would ever buy the player. Similarly, one turns a package over to an overnight delivery service without feeling a need to know anything about the particular kinds of vehicles or routes the service will use. Confidence that the package will be delivered tomorrow is the only concern.

In the computer world, abstraction appears in countless ways. The general ability of sequential circuits to remember state is abstracted into particular, easy-to-describe modules called *registers*. Programs are designed to hide details of their representation of complex data structures and details of which other programs they call. Users expect easy-to-use, button-pushing application interfaces such as computer

It must be remembered that there is nothing more difficult to plan, more doubtful of success, nor more dangerous to manage than the creation of a new system. For the initiator has the enmity of all who would profit by the preservation of the old institutions and merely lukewarm defenders in those who would gain by the new ones.

— Niccolò Machiavelli, *The Prince* (1513, published 1532; Tr. by Thomas G. Bergin, Appleton-Century-Crofts, 1947)

games, spreadsheet programs, or Web browsers that abstract incredibly complex underpinnings of memory, processor, communication, and display management.

The goal of minimizing interconnections among modules may be defeated if unintentional or accidental interconnections occur as a result of implementation errors or even well-meaning design attempts to sneak past modular boundaries in order to improve performance or meet some other requirement. Software is particularly subject to this problem because the modular boundaries provided by separately compiled subprograms are somewhat soft and easily penetrated by errors in using pointers, filling buffers, or calculating array indices. For this reason, system designers prefer techniques that enforce modularity by interposing impenetrable walls between modules. These techniques ensure that there can be no unintentional or hidden interconnections. Chapters 4 and 5 develop some of these techniques for enforcing modularity.

Well-designed and properly enforced modular abstractions are especially important in limiting the impact of faults because they control propagation of effects. As we shall see when we study fault tolerance in Chapter 8 [on-line], modules are the units of fault containment, and the definition of a failure is that a module does not meet its abstract interface specifications.

Closely related to abstraction is an important design rule that makes modularity work in practice:

---

#### The robustness principle

*Be tolerant of inputs and strict on outputs.*

---

This principle means that a module should be designed to be liberal in its interpretation of its input values, accepting them even if they are not within specified ranges, if it is still apparent how to sensibly interpret them. On the other hand, the module should construct its outputs conservatively in accordance with its specification—if possible making them even more accurate or more constrained than the specification requires. The effect of the robustness principle is to tend to suppress, rather than propagate or even amplify, noise or errors that show up in the interfaces between modules.

The robustness principle is one of the key ideas underlying modern mass production. Historically, machinists made components that were intended to mate by machining one of the components and then machining a second component to exactly fit against or into the first one, a technique known as *fitting*. The breakthrough came with the realization that if one specified *tolerances* for components and designed

We are faced with an insurmountable opportunity.

— Pogo (Walt Kelley)

each component to mate with any other component that was within its specified tolerance, then it would be possible to modularize and speed up manufacturing by having interchangeable parts. Apparently, this concept was first successfully applied in an 1822 contract to deliver rifles to the United States Army. By the time production lines for the Model T automobile were created, Henry Ford captured the concept in the aphorism, “In mass production there are no fitters.”

The robustness principle plays a major role in computer systems. It is particularly important in human interfaces, network protocols, and fault tolerance, and, as [Section 1.4](#) of this chapter explains, it forms the basis for digital logic. At the same time, a tension exists between the robustness principle and another important design principle:

---

#### The safety margin principle

---

*Keep track of the distance to the cliff, or you may fall over the edge.*

---

When inputs are not close to their specified values, that is usually an indication that something is starting to go wrong. The sooner that something going wrong can be noticed, the sooner it can be fixed. For this reason, it is important to track and report out-of-tolerance inputs, even if the robustness principle would allow them to be interpreted successfully.

Some systems implement the safety margin principle by providing two modes of operation, which might be called “shake-out” and “production”. In shake-out mode, modules check every input carefully and refuse to accept anything that is even slightly out of specification, thus allowing immediate discovery of problems and of programming errors near their source. In production mode, modules accept any input that they can reasonably interpret, in accordance with the robustness principle. Carefully designed systems blend the two ideas: accept any reasonable input but report any input that is beginning to drift out of tolerance so that it may be repaired before it becomes completely unusable.

### 1.3.3 Layering

Systems that are designed using good abstractions tend to minimize the number of interconnections among their component modules. One powerful way to reduce module interconnections is to employ a particular method of module organization known as *layering*. In designing with layers, one builds on a set of mechanisms that is already complete (a lower layer) and uses them to create a different complete set of mechanisms (an upper layer). A layer may itself be implemented as several modules,

There is no such thing as a small change to a large system.

— **systems folklore, source lost in the mists of time**

but as a general rule, a module of a given layer interacts only with its peers in the same layer and with the modules of the next higher and next lower layers. That restriction can significantly reduce the number of potential intermodule interactions in a big system.

Some of the best examples of this approach are found in computer systems: an interpreter for a high-level language is implemented using a lower-level, more machine-oriented, language. Although the higher-level language doesn't allow any new programs to be expressed, it is easier to use, at least for the application for which it was designed.

Thus, nearly every computer system comprises several layers. The lowest layer consists of gates and memory cells, upon which is built a layer consisting of a processor and memory. On top of this layer is built an operating system layer, which acts as an augmentation of the processor and memory layer. Finally, an application program executes on this augmented processor and memory layer. In each layer, the functions provided by the layer below are rearranged, repackaged, reabstracted, and reinterpreted as appropriate for the convenience of the layer above. As will be seen in Chapter 7 [on-line], layers are also the primary organizing technique of data communication networks.

Layered design is not unique to computer systems and communications. A house has an inner structural layer of studs, joists, and rafters to provide shape and strength, a layer of sheathing and drywall to keep the wind out, a layer of siding, flooring and roof tiles to make it watertight, and a cosmetic layer of paint to make it look good. Much of mathematics, particularly algebra, is elegantly organized in layers (in the case of algebra, integers, rationals, complex numbers, polynomials, and polynomials with polynomial coefficients), and that organization provides a key to deep understanding.

#### 1.3.4 Hierarchy

The final major technique for coping with complexity also reduces interconnections among modules but in a different, specialized way. Start with a small group of modules, and assemble them into a stable, self-contained subsystem that has a well-defined interface. Next, assemble a small group of subsystems to produce a larger subsystem. This process continues until the final system has been constructed from a small number of relatively large subsystems. The result is a tree-like structure known as a *hierarchy*. Large organizations such as corporations are nearly always set up this way, with a manager responsible for only five to ten employees, a higher-level manager responsible for five to ten managers, on up to the president of the company, who may

The first 80 percent of a project takes 80 percent of the effort.

The last 20 percent takes another 80.

— source unknown

supervise five to ten vice presidents. The same thinking applies to armies. Even layers can be thought of as a kind of degenerate one-dimensional hierarchy.

There are many other striking examples of hierarchy, ranging from microscopic biological systems to the assembly of Alexander's empire. A classic paper by Herbert Simon, "The architecture of complexity" [Suggestions for Further Reading 1.4.3], contains an amazing range of such examples and offers compelling arguments that, under evolution, hierarchical designs have a better chance of survival. The reason is that hierarchy constrains interactions by permitting them only among the components of a subsystem. Hierarchy constrains a system of  $N$  components, which in the worst case might exhibit  $N \times (N - 1)$  interactions, so that each component can interact only with members of its own subsystem, except for an interface component that also interacts with other members of the subsystem at the next higher level of hierarchy. (The interface component in a corporation is called a "manager"; in an army it is called the "commanding officer"; for a program it is called the "application programming interface".) If subsystems have a limit of, say, 10 components, this number remains constant no matter how large the system grows. There will be  $N/10$  lowest level subsystems,  $N/100$  next higher level subsystems, and so on, but the total number of subsystems, and thus the number of interactions, remains proportional to  $N$ . Analogous to the way that modularity reduces the effort of debugging, hierarchy reduces the number of potential interactions among modules from square-law to linear.

This effect is most strongly noticed by the designer of an individual module. If there are no constraints, each module should in principle be prepared to interact with every other module of the system. The advantage of a hierarchy is that the module designer can focus just on interactions with the interfaces of other members of its immediate subsystem.

### 1.3.5 Putting it Back Together: Names Make Connections

The four techniques for coping with complexity—modularity, abstraction, layering, and hierarchy—provide ways of dividing things up and placing the resulting modules in suitable relation one to another. However, we still need a way of connecting those modules. In digital systems, the primary connection method is that one module *names* another module that it intends to use. Names allow postponing of decisions, easy replacement of one module with a better one, and sharing of modules. Software uses names in an obvious way. Less obviously, hardware modules connected to a bus also use names for interconnection—addresses, including bus addresses, are a kind of name.

Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.

— Douglas Hofstadter: *Gödel, Escher, Bach: An Eternal Golden Braid* (1979)

In a modular system, one can usually find several ways to combine modules to implement a desired feature. The designer must at some point choose a specific implementation from among many that are available. Making this choice is called *binding*. Recalling that the power of modularity comes from the ability to replace an implementation with a better one, the designer usually tries to maintain maximum flexibility by delaying binding until the last possible instant, perhaps even until the first instant that the feature is actually needed.

One way to delay binding is just to name a feature rather than implementing it. Using a name allows one to design a module as if a feature of another module exists, even if that feature has not yet been implemented, and it also makes it mechanically easy to later choose a different implementation. By the time the feature is actually invoked, the name must, of course, be bound to a real implementation of the other module. Using a name to delay or allow changing a binding is called *indirection*, and it is the basis of a design principle:

---

#### Decouple modules with indirection

*Indirection supports replaceability.*

---

A folk wisdom version of this principle, attributed to computer scientist David Wheeler of the University of Cambridge, exaggerates the power of indirection by suggesting that “any problem in a computer system can be solved by adding a layer of indirection.” A somewhat more plausible counterpart of this folk wisdom is the observation that any computer system can be made faster by removing a layer of indirection.

When a module has a name, several other modules can make use of it by name, thereby sharing the design effort, cost, or information contained in the first module. Because names are a cornerstone element of modularity in digital systems, Chapters 2 and 3 are largely about the design of naming schemes.

---

## 1.4 COMPUTER SYSTEMS ARE THE SAME BUT DIFFERENT

As we have repeatedly suggested, there is an important lesson to be drawn from the wide range of examples used up to this point to illustrate system problems. Certain common problems show up in all complex systems, whatever their field. Emergent properties, propagation of effects, incommensurate scaling, and trade-offs are considerations in activities as diverse as space station design, management of the economy,

A system is never finished being developed until it ceases to be used.

— attributed to Gerald M. Weinberg

the building of skyscrapers, gene-splicing, petroleum refineries, communication satellite networks, and the governing of India, as well as in the design of computer systems. Furthermore, the techniques that have been devised for coping with complexity are universal. Modularity, abstraction, layering, and hierarchy are used as tools in most fields that deal with complex systems. It is therefore useful for the computer system designer to investigate systems from other fields, both to gain additional perspective on how system problems arise and to discover specific techniques from other fields that may also apply to computer systems. Stated briefly, we conclude that *computer systems are the same as all other systems*.

But there is one problem with that conclusion: it is wrong. There are at least two significant ways in which computer systems differ from every other kind of system with which designers have experience:

- *The complexity of a computer system is not limited by physical laws.*
- *The rate of change of computer system technology is unprecedented.*

These two differences have an enormous impact on complexity and on ways of coping with it.

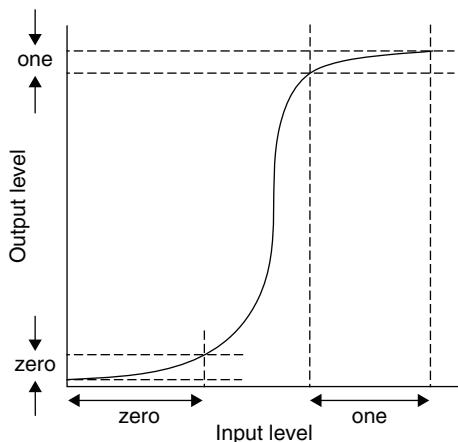
#### 1.4.1 Computer Systems have no Nearby Bounds on Composition

Computer systems are mostly digital, and they are controlled by software. Each of these two properties separately leads to relaxations of what, in other systems, would be limits on complexity arising from physical laws.

Consider first the difference between analog and digital systems. All analog systems have the engineering limitation that each component of the system contributes noise. This noise may come from the environment in the form of, for example, vibration or electromagnetic radiation. Noise may also appear because the component's physical behavior does not precisely follow any tractable model of operation: the pile of rocks that a civil engineer specifies to go under a bridge abutment does not obey a simple deformation model; a resistor in an electronic circuit generates random noise whose level depends on the temperature. When analog components are composed into systems, the noise from individual components accumulates (if the noise sources are statistically independent, the noise may accumulate only slowly but it still accumulates). As the number of components increases, noise will at some point dominate the behavior of the system. (This analysis applies to systems designed by human engineers.

I was to learn later in life that we tend to meet any new situation by reorganisation; and what a wonderful method it can be for creating the illusion of progress while producing confusion, inefficiency and demoralisation.

— shortened version of an observation by Charlton Ogburn, “Merrill’s Marauders: The truth about an incredible adventure”, *Harper’s Magazine* (January 1957). Widely but improbably misattributed to Petronius Arbiter (ca. A.D. 60)



**FIGURE 1.3**

How gain and non-linearity of a digital component restore levels. The input level and output level span the same range of values, but the range of accepted inputs is much wider than the range of generated outputs.

In contrast, digital systems are noise-free; complexity can therefore grow without any constraint of a bound arising from noise. The designers of digital logic use a version of the *robustness principle* known as the *static discipline*. This discipline is the primary source of the magic that seems to surround digital systems. The static discipline requires that the range of analog values that a device accepts as meaning the digital value ONE (or ZERO) be wider than the range of analog values that the device puts out when it means digital ONE (or ZERO). This discipline is an example of being tolerant of inputs and strict on outputs.

Digital systems are, at some lower level, constructed of analog components. The analog components chosen for this purpose are non-linear, and they have gain between input and output. When used appropriately, non-linearity allows inputs to have a wide tolerance, and gain ensures that outputs stay within narrow specifications, as shown in Figure 1.3. Together they produce the property of digital circuits called level restoration or regeneration. Regenerated signal levels appear at the output of every digital component, whatever their level of granularity: a gate, a flip-flop, a memory chip, a processor, or a complete computer system. Regenerated levels create clean interfaces that allow one subsystem to be connected to the next with

Natural biological, thermodynamic, and macroeconomic systems, composed of billions of analog components, somehow use hierarchy, layering, abstraction, and modularity to operate despite noise, but they are so complex that we do not understand them well enough to adopt the same techniques.)

Noise thus provides a limit on the number of analog components that a designer can usefully compose or on the number of stages that a designer can usefully cascade. This argument applies to any engineered analog system: a bridge across a river, a stereo, or an airliner. It is the reason a photocopy of a photocopy is harder to read than the original. There may also be other limits on size (arising from the strength of materials, for example), but noise is always a limit on the complexity of analog systems.

The probability of failure of a system tends to be proportional to the confidence that its designer has in its reliability.

— systems folklore, source lost

confidence. Unlike the civil engineer's pile of rocks, a logic gate performs exactly as its designer intends.

The static discipline and level restoration do *not* guarantee that devices with digital inputs and outputs never make mistakes. Any component can fail. Or an input signal that is intended to be a ONE may be so far out of tolerance that the receiving component accepts it as a ZERO. When that happens, the output of the component that accepted that value incorrectly is likely to be wrong, too. The important consequence is that digital components make big mistakes, not little ones, and as we shall see when we reach the chapter on fault tolerance, big mistakes are relatively easy to detect and handle.

If a signal does not accumulate noise as it goes through a string of devices, then noise does not limit the number of devices one can string together. In other words, noise does not constrain the maximum depth of composition for digital systems. Unlike analog systems, digital systems can grow in complexity until they exceed the ability of their designers to understand them. As of 2009, processor chips contain over two billion transistors, far more than any analog chip. No airliner has nearly that many components—except in its on-board computers.

The second reason composition has no nearby bounds is that computer systems are controlled by software. Bad as the contribution to complexity from the static discipline may be, the contribution from software turns out to be worse. Hardware is at least subject to *some* physical limits—the speed of light, the rate of settling of signals in real semiconductor materials, unwanted electrical coupling between adjacent components, the rate at which heat can be removed, and the space that it occupies. Software appears to have no physical limits whatever beyond the availability of memory to store it and processors to execute it. As a result, composition of software can go on as fast as people can create it. Thus one routinely hears of operating systems, database systems, and even word processors consisting of more than 10 million program statements.

In principle, abstraction can help control software composition by hiding implementation beneath module interfaces. The problem is that most abstractions are, in reality, slightly “leaky” in that they don't perfectly conceal the underlying implementation. A simple example of leakiness is addition of integers: in most implementations, the addition operation perfectly matches the mathematical specification as long as the result fits in the available word size, but if the result is larger than that, the resulting overflow becomes a complication for the programmer. Leakiness, like noise in analog systems, accumulates as the number of software modules grows. Unlike noise, it accumulates in the form of complexity, so the lack of physical constraints on

The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair.

— Douglas Adams, *Mostly Harmless (Hitchhiker's Guide to the Galaxy V)* (1993)

software composition remains a fundamental problem. It is, therefore, mechanically easy to create a system with complexity that is far beyond the ability of its designers to understand. And since it is easy, it happens often, and sometimes with disastrous results.\*

Between the absence of a noise-imposed limit on composition of digital hardware and very distant physical limits on composition of software, it is too easy for an unwary designer to misuse the tools of modularity, abstraction, layering, and hierarchy to include still more complexity. This phenomenon is quite unknown in the design of bridges and airliners. *In contrast with other systems, computer systems allow composition to a depth whose first limit is the designer's ability to understand.* Unfortunately, this lack of nearby natural, physical bounds on depth of composition tempts designers to build more complex systems. If nature does not impose a nearby limit on composition, the designer must self-impose a limit. Since it can be hard to say no to a reasonable-sounding feature, features keep getting added. Therein lies the fate of too many computer system designs.

### 1.4.2 $d(\text{technology})/dt$ is Unprecedented

For reasons partly explained by Sidebar 1.6, during the last 35 years the cost of the digital hardware used for computation and communication has dropped an average of about 30% each year. This rate of change means that just two years' passage of time has been enough to allow technology to cut prices in half, and in seven or eight years it has led to a drop in prices by a factor of 10. Some components have experienced even greater rates of improvement. Figure 1.4 shows the cost of magnetic disk storage over a 25-year span. During that time, disk prices have actually dropped by a factor of 10 roughly every five years, so disk prices have dropped nearly 60% each year. Disk experts project a similar rate of improvement for at least another few years. Their projection seems relatively safe, since no major roadblocks have been reported by development laboratories that are already working on the next rounds of magnetic recording technology. Similar charts apply to random access memory, processor cost, and the speed of optical fiber transmission.

This rapid change of technology has created a substantial difference between computer systems and other engineering systems. Since complex systems can take several years to build, by the time a computer system is ready for delivery, the

---

\*The terminology “leaky” is apparently due to software developer Joel Spolsky.

Structural engineering is the art of modeling materials we do not wholly understand, into shapes we cannot precisely analyse so as to withstand forces we cannot properly assess, in such a way that the public has no reason to suspect the extent of our ignorance.

— A. R. Dykes, Scottish Branch, Institution of Structural Engineers (1946)

**Sidebar 1.6 Why Computer Technology has Improved Exponentially with Time** Popular media frequently use the term “exponential” to describe the explosive rate of improvement of computer technology. Stephen Ward has pointed out that there is a good reason this adjective is appropriate: computer technology appears to be the rare engineering discipline in which the technology being improved is routinely employed to improve the technology. People building airplanes, bridges, skyscrapers, and chemical plants rarely, if ever, have this opportunity.

For example, the performance of a microprocessor is determined at least in part by the cleverness of its layout, which in turn is limited by the time available to use computer-assisted layout tools that can take advantage of lithography advances. If Intel, through improved layout, makes a version of the Pentium that is twice as fast, as soon as that new Pentium is available, it will be used as the processor to make the layout tools for the next Pentium run twice as fast; the next design can benefit from twice as much computation in its layout. This effect is probably one of the drivers of Moore’s law, which predicts an exponential increase in component count on chips with a doubling time of 18 months [Suggestions for Further Reading 1.6.1].

If indeed the rate at which we can improve our technology is proportional to the quality of the technology itself, we can express this idea as

$$\frac{d(\text{technology})}{dt} = K \times \text{technology}$$

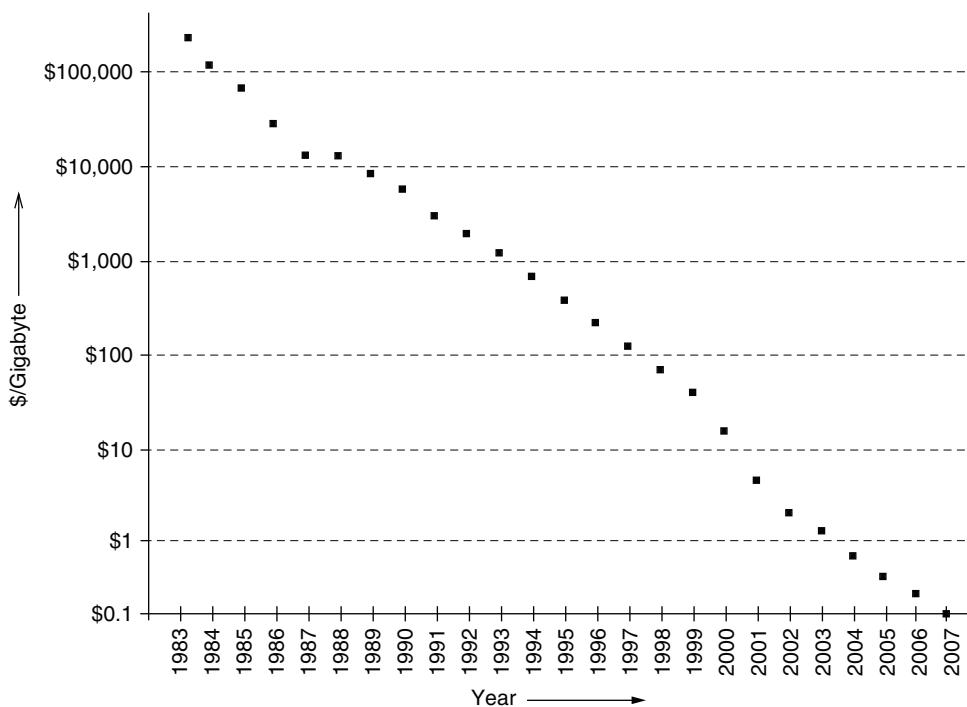
which has an exponential solution,

$$\text{technology} = e^{Kt}$$

The actual situation is, of course, far more complicated than that equation suggests, but all equations that even remotely resemble that form, in which technology’s rate of growth is some positive function of its current state, have growing exponentials in their solution.

In the real world, exponentials must eventually hit some limit. In hardware there are fairly clear fundamental physical limits to exponential growth, such as the uncertainty principle, the minimum energy required to switch a gate, and the rate at which heat can be removed from a device. The interesting part is that it isn’t obvious which one is going to become the roadblock, or when. Thus far, engineering ingenuity in exploiting trade-offs has postponed the day of reckoning. For software, similar limits on exponential growth must exist, but their nature is not at all clear.

More to the immediate point, virtually every improvement in computer and communications technology—whether faster chips, better Internet routing algorithms, more effective prototyping languages, better browser interfaces, faster compilers, bigger disks, or larger RAM—is immediately put to work by everyone who is working on faster chips, better Internet routing algorithms, more effective prototyping languages, better browser interfaces, faster compilers, bigger disks, or larger RAM. Computer system designers live inside a giant feedback system that, at least for the moment, is enjoying exponential solutions.

**FIGURE 1.4**

Magnetic disk price history and projection, 1983–2007.

ground rules under which it was originally designed have shifted. Incommensurate scaling typically means that the designer must adjust for strains when any system parameter changes by a factor of 2, because not all of the components scale up (or down) by the same proportion. More to the point, a whole new design is usually needed when any system parameter changes by a (decimal) order of magnitude. This rule of thumb about strains caused by parameter changes gives us our next design principle:

---

**The incommensurate scaling rule**

*Changing any system parameter by a factor of 10 usually requires a new design.*

---

If you design it so that it can be assembled wrong, someone will assemble it wrong.

— Edward A. Murphy, Jr. (paraphrase of the original Murphy's law, 1949; see sidebar 2.5)

This rule, when combined with the observed rate of change of technology, means that by the time a newly designed computer system is ready for delivery it may have already needed two rounds of adjustment and be ready for a complete redesign. Even if the designer has tried to predict the impact of technology change, crystal balls are at best cloudy. Worse, during the development of the system, things may run an order of magnitude slower than they will when the system is finished, the code and data don't fit in the available address space, or perhaps the data has to be partitioned across several hard disks instead of nicely fitting on one. One can compensate for each of these problems, but each such compensation absorbs intellectual resources and contributes complexity to the development process.

Even without those adjustments or redesign, the original plan was probably already a new design. A bridge (or airplane) may have a modest number of things that are different from the previous one, but a civil (or aeronautical) engineer almost always ends up designing something that is only a little different from some previous bridge (or airplane). In the case of computer systems, ideas that were completely unrealistic a year or two ago can become mainstream in no time, so the computer system designer almost always ends up designing something that is significantly different from the previous computer system. This difference makes deep analysis of previous designs more rewarding for civil and aeronautical engineers than for computer system designers, and also usually means that in computer systems there hasn't been time to discover and iron out most of the mistakes of the previous design before going on to the next major revision. Those mistakes can contribute strongly to complexity.

Because technology has improved so rapidly, the field of computer system design tends to place much less emphasis on detailed performance analysis and fine-tuning than do most other engineering endeavors. Where an electric power generation system may benefit dramatically from a new steam turbine that improves energy transfer by 1%, a needed 20% improvement in performance of a computer system can usually be obtained just by waiting four months for the next round of hardware product announcements. If a proposal to rewrite an application to obtain that same improvement would require a year of work, it is probably more cost-effective to just wait for technology change to solve the problem. Put another way, rapidly improving technology means that brute-force solutions (buy more memory, wait for a faster processor, use a simpler algorithm) are often the right approach in computer systems, whereas in other systems they may be unthinkable. The owner of the railroad through the canyon probably would not view as economically reasonable a proposal to blast the canyon wider and install a second track. Even if the resources were available, the environmental impact would be a deterrent.

This “telephone” has too many shortcomings to be seriously considered as a means of communication. The device is inherently of no value to us.

— frequently attributed to an 1876 Western Union internal memo, but there is no evidence of this memo and it is probably a myth.

A second major consequence of the rapid rate of change of technology in computer systems is that usability, and related qualities that go under the label “human engineering”, of computer systems is always ragged. It takes years of trial and error to make systems usable, friendly, and forgiving, but by the time one level of computer technology has been tamed, a new level of computer technology opens the possibilities of many new features at the same cost, or of providing the previous features more cheaply to a vast new audience of unprepared users.

Similarly, legal and judicial processes take decades to come to grips with new issues, as people debate the wisdom of various policies, discover abuses, and explore alternative remedies. In the face of rapidly changing computer system technology, these processes fall far behind, delaying resolution of such concerns as how to reward innovative software ideas, or what rules should protect information stored in computers, and adding uncertainty of requirements to the burden of the computer system designer.\*

Finally, modern high-speed communications with global reach have greatly accelerated the rate at which people discover that a new technology is useful and adopt it. Where it took several decades for electricity and the telephone to move from curiosities to widespread use, recent innovations such as digital cameras and DVDs have swept their markets in less than a decade, and a single mention of a previously obscure World Wide Web site on CNN or in *Newsweek* magazine can cause that site to be suddenly overwhelmed with millions of hits per day. More generally, newly viable applications, such as peer-to-peer file sharing, can change the shape of the workload on existing systems practically overnight.

Thus, the study of computer systems involves telescoping of the usual processes of planning, examining requirements, tailoring details, and integrating with users and society. This telescoping leads to the delivery of systems that have rough edges and without the benefit of the cleverest thought. People who build airplanes and bridges do not have to face these problems. Such problems can be viewed either as a frustrating difficulty or as an exciting challenge, depending on one’s perspective.

## 1.5 COPING WITH COMPLEXITY II

Modest physical limits in hardware and very distant physical limits in software together give us the opportunity to create systems of unimaginable—and unmanageable—complexity, and the rapid pace of technology change tempts designers to deliver

\*Lawrence Lessig provides a good analysis of the interactions of law, society, and computer technology in *Code: and Other Laws of Cyberspace* [Suggestions for Further Reading 1.1.4].

Books will soon be obsolete in the public schools. . . . It is possible to teach every branch of human knowledge with the motion picture. Our school system will be completely changed inside of ten years.

— Thomas A. Edison, as quoted in the *New York Dramatic Mirror* (July 9, 1913)

systems using new and untested ground rules. These two effects amplify the complexity of computer systems when compared with systems from other engineering areas. Thus, computer system designers need some additional tools to cope with complexity.

### 1.5.1 Why Modularity, Abstraction, Layering, and Hierarchy aren't Enough

Modularity, abstraction, layering, and hierarchy are a major help, but by themselves they aren't enough to keep the resulting complexity under control. The reason is that all four of those techniques assume that the designer understands the system being designed. In the real, fast-changing world of computer systems, it is hard to choose

- the *right* modularity from a sea of plausible alternative modularities.
- the *right* abstraction from a sea of plausible alternative abstractions.
- the *right* layering from a sea of plausible alternative layerings.
- the *right* hierarchy from a sea of plausible alternative hierarchies.

Although some design principles are available, they are far too few, and the only real guidance comes from experience with previous systems.

As might be expected, designers of computer systems have developed and refined at least one additional technique to cope with complexity. Designers of other kinds of systems use this technique as well, but they usually do not consider it to be so fundamental to success as it is for computer systems, probably because the technique is particularly feasible with software. It is a development process called *iteration*.

### 1.5.2 Iteration

The essence of iteration is to start by building a simple, working system that meets only a modest subset of the requirements and then evolve that system in small steps to gradually encompass more and more of the full set of requirements. The idea is that small steps can help reduce the risk that complexity will overwhelm a system design. Having a working system available at all times helps provide assurance that something can be built and provides on-going experience with the current technology ground rules as well as an opportunity to discover and fix bugs. Finally, adjustments for technology changes that arrive during the system development are easier to incorporate as part of one or more of the iterations. When you see a piece of software identified as "release 5.4", that is usually an indication that the vendor is using iteration.

Successful iteration requires considerable foresight. That foresight involves several elements, two of which we identify as design principles:

I think there is a world market for maybe five computers.

— Frequently claimed to be said by Thomas J. Watson, Sr., chairman of IBM, in a 1943 talk, but there is little evidence that it is anything but a legend.

- First of all,
- 

### Design for iteration

*You won't get it right the first time, so make it easy to change.*

---

Document the assumptions behind the design so that when the time comes to change the design you can more easily figure out what else has to change. Expect not only to modify and replace modules, but also to remodularize as the system and its requirements become better understood.

- *Take small steps.* The purpose is to allow discovery of both design mistakes and bad ideas quickly, so that they can be changed or removed with small effort and before other parts of the system in later iterations start to depend on them and they effectively become unchangeable. Systems under active development may be subjected to a complete system rebuild every day because the rebuilding process invokes a large number of checks and tests that can reveal implementation mistakes, while the changes that caused the mistakes are fresh in the minds of the implementers.
  - *Don't rush.* Even though individual steps may be small, they must still be well planned. In most projects, the temptation is to rush to implementation. With iterative design, that temptation can be stronger, and the designer must make sure that the design is ready for the next step.
  - *Plan for feedback.* Include as part of the design both feedback paths and positive incentives to provide feedback. Testers, installers, maintainers, and users of the system can provide much of the information needed to refine it. Alpha testing (“we're not at all sure this even works”) and beta testing (“seems to work, use at your own risk”) are common examples, and many vendors encourage users to report details of problems and transcripts of failures by e-mail. A well-designed system will provide many such feedback schemes at all levels.
  - *Study failures.* An important goal is to learn from failures rather than assign blame for them. Incentives must be carefully designed to ensure that feedback about failures is not ignored or even suppressed by people fearful of being blamed. Then, having found the apparent cause of a failure,
- 

### Keep digging

*Complex systems fail for complex reasons.*

---

Computers in the future may weigh no more than 1.5 tons.

— *Popular Mechanics* (March 1949)

Continue looking for other contributing or more basic causes. Working systems often work for reasons that aren't well understood. It is common to find that a new release of a system reveals a bug that has actually been in the system for a long time but has never mattered until now. Much can be learned by figuring out why it never mattered. It can also be useful to explore the mindset of the designers to understand what allowed them to design a system that could fail in this way.\* Similarly, don't ignore unexplained behavior. If the feedback reports something that now seems not to be a problem or to have gone away, it is probably a sign that something is wrong rather than that the system magically fixed itself.

Iteration sounds like a straightforward technique, but several obstacles tend to interfere with it. The main obstacle is that as a design evolves through a series of iterations, a risk of losing conceptual integrity arises. That risk suggests that the overall plan for the initial, simplest version of the system must accommodate all of the iterations needed to reach the final version (thus the need for foresight). Someone must constantly be on guard to make sure that the overall design rationale remains clear despite changes made during iteration.

In most organizations, good news (e.g., a major piece of the system is working ahead of schedule) flows rapidly throughout the organization, but bad news (e.g., an important module isn't working yet) often gets confined to the part of the organization that discovers it, at least until it can fix the problem and report good news. This phenomenon, the *bad-news diode*, can prevent realization that changing a different part of the system is more appropriate.

A related problem is that when someone finally realizes that the modularity is wrong, it can be hard to change, for two reasons. First, the *unyielding foundations rule* (see page 20) comes into play. Changing modularity by definition involves changing more than one module, and sometimes several. Second, designers who have invested time and effort in developing a module that, from their point of view, is doing what was intended can be reluctant to see this time and effort lost in a rework. Simply put, to change modularity one must deal with both committed components and committed designers.

---

\*The idea of learning from failure and the observation that complex systems fail for complex reasons are the themes of a fascinating book by Henry Petroski, *Design Paradigms: Case Histories of Error and Judgment in Engineering* [Suggestions for Further Reading 1.2.3].

Based on extensive financial and market analysis, it's projected that no more than five thousand of the new Haloid machines will sell. . . . Model 914, has no future in the office copying market.

— Consulting firm Arthur D. Little's report to IBM on the prospects for xerographic copying machines (1959)

A longer-term risk of iteration sometimes shows up when the initial design is both simple and successful. Success can lead designers to be overconfident and to be too ambitious on a later iteration. Technology has improved in the time since deployment of the initial version of the system and feedback has suggested lots of new features. Each suggested feature looks straightforward by itself, and it is difficult to judge how they might interact. The result is often a disastrous overreaching and consequent failure that is so common that it has a name: the *second-system effect*.

Iteration can be thought of as applying modularity to the management of the system design and implementation process. It thus takes us into the realm of management techniques, which are not directly addressed in this book.\*

### 1.5.3 Keep it Simple

Remarkably, one of the most effective techniques in coping with complexity is also one that is most difficult to apply: *simplicity*. As Section 1.4.1 explained, computer systems lack natural physical limits to curb their complexity, so the designer must impose limits; otherwise the designer risks being overwhelmed.

The problem with the apparently obvious advice to keep it simple is that

- previous systems give a taste of how great things could be if more features were added.
- the technology has improved so much that cost and performance are not constraints.
- each of the suggested new features has been successfully demonstrated somewhere.
- none of the exceptions or other complications seems by itself to be especially hard to deal with.
- there is fear that a competitor will market a system that has even more features.
- among system designers, arrogance, pride, and overconfidence are more common than clear awareness of the dangers of complexity.

These considerations make it hard to say “no” to any one requirement, feature, exception, or complication. It is their cumulative impact that produces the complexity explosion illustrated in Figure 1.1. The system designer must keep this cumulative

\*An excellent book on the subject of system development, by a veteran designer, is Frederick P. Brooks Jr., *The Mythical Man-Month* [Suggestions for Further Reading 1.1.3]. Another highly recommended reading is the Alan Turing Award lecture by Fernando J. Corbató, “On building systems that will fail” [Suggestions for Further Reading 1.5.3].

There is no reason anyone would want a computer in their home.

— Kenneth Olsen, president of Digital Equipment Corporation (1977)

impact in mind at all times. The bottom line is that a computer system designer's most potent weapon against complexity is the ability to say, "No. This will make it too complicated."

As we proceed to study specific computer system engineering topics, we shall make much use of a particular kind of simplicity, to the extent that it is yet another design principle:

---

#### Adopt sweeping simplifications

*So you can see what you are doing.*

---

Each topic area will explicitly introduce one or more sweeping simplifications. The reason is that they allow the designer to make compelling arguments for correctness, they make detail irrelevant, and they make clear to all participants exactly what is going on. They will turn out to be one of our best hopes for keeping control of complexity.

---

## WHAT THE REST OF THIS BOOK IS ABOUT

This chapter has introduced some basic ideas that underlie the study of computer systems. In the course of building on these basic ideas, the ensuing chapters explore a series of system engineering topics in the light of three recurring themes:

- The pervasive importance of modularity
- Principle-based system design
- Making systems robust and resilient

Modularity appears in each engineering topic either as one of the goals of that topic or as one of its design cornerstones. Words from chapter titles suggest this theme. *Abstractions and layering* are particular ways to build on modularity. *Naming* is a fundamental mechanism for interconnecting and replacing modules. *Clients and services* and *virtualization* are two ways of enforcing modularity. *Networks* are built on a foundation of modularity. In *fault tolerance*, the module is the unit that limits the extent of failure. *Atomicity* is an exceptionally robust form of modularity that the designer can exploit to obtain *consistency*. Finally, *protection of information* involves further strengthening of modular walls.

The second theme, principle-based system design, has already emerged, both in explicit mention of several principles and in the list of *design principles* on the inside front cover. These principles capture, in brief phrases, widely applicable nuggets of wisdom that have been developed by generations of computer system designers. Later chapters apply these general principles and also introduce additional design principles that are more specific to particular engineering areas. Even with these principles in mind, it is often difficult to offer a precise recipe for design. Therefore throughout the text the reader will find a second form of captured wisdom in the form of several design *hints* that encode

rationales for making trade-offs.\* Together, the principles and hints suggest that computer system design, though for the most part not based on mathematical theories, is also not completely *ad hoc*: it is actually based on sound principles derived from experience and analysis of both successful and failed systems. The reader who understands and absorbs these principles and hints will have learned much of what this book has to say.

The third theme, making systems robust and resilient, has also already emerged, both in the statement of the *robustness principle* and with the idea that modularity, by limiting interconnections, can help control propagation of effects. The terms *robustness* and *resilience* are informal and overlapping descriptions of a general goal of design: that a system should not be sensitive to modest, long-term shifts in its environment (usually called robustness) and that it should continue operating correctly in the face of transient adversity (usually called resilience). Each succeeding chapter introduces at least one progressively stronger way to make a system more robust and resilient. Thus, the chapter on naming shows how indirection of names can make systems less fragile. Then, the chapters on clients and services and on virtualization demonstrate how to enforce modularity to limit the effects of mistakes and accidents. The chapter on networks introduces techniques that provide reliable communications despite communication failures. The chapter on fault tolerance then generalizes those techniques to make entire systems resilient, even though they contain faulty components. The chapters on atomicity and consistency apply fault tolerance techniques to the particular problem of maintaining the integrity of stored data, despite concurrent activity and in the face of software and hardware failures. Finally, the chapter on protecting information introduces techniques to limit the impact of malicious adversaries who would deliberately steal, modify, or deny access to information.

## EXERCISES

**1.1** True or false? Explain: modularity reduces complexity because

- A. It reduces the effect of incommensurate scaling.
- B. It helps control propagation of effects.

*1994-1-3d and 1995-1-1e*

**1.2** True or false? Explain: hierarchy reduces complexity because

- A. It reduces the size of individual modules.
- B. It cuts down on the number of interconnections between elements.
- C. It assembles a number of smaller elements into a single larger element.
- D. It enforces a structure on the interconnections between elements.
- E. All of the above.

*1994-1-3c and 1999-1-02*

\*Many, if not all, of the hints were originally described by Butler Lampson in his paper “Hints for computer system design” [Suggestions for Further Reading 1.5.4].

- 1.3** If one created a graph of personal friendships, one would have a hierarchy.  
True or false?

*1995-1-1b*

- 1.4** Which of the following is usually observed in a complex computer system?
- A.** The underlying technology has a high rate of change.
  - B.** It is easy to write a succinct description of the behavior of the system.
  - C.** It has a large number of interacting features.
  - D.** It exhibits emergent properties that make the system perform better than envisioned by the system's designers.

*2005-1-1*

- 1.5** Ben Bitdiddle has written a program with 16 major modules of code. Each module contains several procedures. In the first implementation of his program, he finds that each module contains at least one call to every other module. Each module contains 100 lines of code.

**1.5a** How long is Ben's program in lines of code?

**1.5b** How many module interconnections are there in his implementation? (Each call from one module to another is an interconnection.)

Ben decides to change the implementation. Now there are four main modules, each containing four submodules in a one-level hierarchy. The four main modules each have calls to all the other main modules, and within each main module, the four submodules each have calls to one another. There are still 100 lines of code per submodule, but each main module needs 100 lines of management code.

**1.5c** How long is Ben's program now?

**1.5d** How many interconnections are there now? Include module-to-module and submodule-to-submodule interconnections.

**1.5e** Was using hierarchy a good decision? Why or why not?

*1996-1-2a...e*

**Additional exercises relating to Chapter 1 can be found in the problem sets beginning on page 425.**

# Interface Stability

Mark Kampe

## Interface Specifications

American History books like to credit Eli Whitney with the invention of manufacturing with interchangeable parts. Actually,

- Whitney's parts weren't all that interchangeable,
- he got the idea from a French gunsmith (Honore Blanc),
- the Swedish clock maker Christopher Polhem did a much better job with clock gears 50 years earlier.

But, be that as it may, interchangeable parts revolutionized, and in fact enabled modern manufacturing. The underlying concept is that there be specifications for every part. If each part is manufactured and measured to be within its specifications, then **any** collection of these parts can be assembled into a working whole. This greatly reduces the cost and difficulty of building a working system out of component parts.

The same principle applies to software. If you want to open a file on a Posix system, you use the **open** system call. There may be a hundred different implementations of open but this doesn't matter because they all conform to the same interface specification. The rewards for this standardization are:

- a. Your work as a programmer is simplified, because you don't have to write your own file I/O routines.
- b. Your program is likely to be easily portable to any Posix compliant system, because they all provide the same file I/O services.
- c. Training time for new programmers is reduced, because everybody already knows how to use the Posix file I/O functions.

## The Criticality of Interfaces in Architecture

A system architecture defines the major components of the system, the functionality of each, and the interfaces between them. Clearly the component interface specifications are a critical element of any architecture. To the extent that these interfaces have been well considered and well defined, it should be possible to (at least semi) independently design and implement those components. In principle, any implementations that satisfy those functional and interface specifications should combine to yield a working system.

The converse is also true. To the extent that interfaces between components were poorly considered or specified, it becomes unlikely that independently developed components will work when combined together, and more likely that subsequent changes to one component will break others.

## The Importance of Interface Stability

We write contracts so that everybody knows what will be expected of them, and what they can expect from the other parties to the agreement. Everybody will depend on you to uphold your obligations under the contract. A software interface specification is a form of contract:

- the contract describes an interface and the associated functionality.
- implementation providers agree that their systems will conform to the specification.
- developers agree to limit their use of the functionality to what is described in the interface specification.

And in return for this they get all the benefits described above.

Suppose that some architectural genius realized that Posix file semantics are too weak to describe the behavior of files in a distributed system, and that this could be solved by adding a new (required) parameter (for a call-back routine) to the Posix open call. What would happen?

- software written to the new semantics would have access to richer behavior, and would function better in cases of node and network failures.
- software written to the new semantics would not work on other Posix compliant systems.
- software written to the old semantics would not work on the new systems.
- customers (knowing nothing about this techno-feud) would be faced with inexplicable failures, and would start complaining to their software and system providers (none of whom were responsible for the change).
- programmers would be confused about which version of open they were using, and would probably get around the problem by writing their own file I/O packages ... a bunch of extra work, but at least it would protect them from future such acts of mischief.

And as a result of this, we would lose all of the benefits described above. When you promise somebody that you will do something (e.g. conform to a specified interface), and they depend on you (e.g. by writing code to that interface), and you don't follow through (e.g. make an incompatible change) ... problems are likely to ensue (e.g. failures, bug reports, product gets a bad reputation, going out of business, etc.).

## Interfaces vs. Implementations

Suppose that someone writes a handy library to efficiently provide reliable communication over an unreliable network, and I decide I want to use it. I download their development kit and start using it, and find it to be great. A few weeks into the project I realize that I need a feature that is not included in their documentation, but after reading their code I discover that the needed feature is actually available. I use it and my product is a great success.

Six months later, they release a new version of their reliable communications library, and my product immediately breaks. After a little debugging I discover that they have changed the undocumented code that I was depending on. It turns out that I had not designed my product to work with their *interfaces*. My product only worked with a particular *implementation* ... and implementations change.

People often put much more work into designing and maintaining their code than to clarifying and maintaining their documentation. This makes it tempting to reverse engineer the interface specifications from a provided implementation. But an interface should exist (and be defined) independently from any particular implementation. Confusing an implementation with an interface usually ends badly!

## Program vs. User Interfaces

Human beings are amazingly robust. We could change the text in dialog boxes, rearrange input forms, add new required input items, and rework all of the menus in a program, and many users would figure out the changes in a few seconds. This inclines many developers to be cavalier in making changes to user interfaces.

Code is nowhere nearly so robust. If I expect my second parameter to be a file name, and you pass me the address of a call-back routine instead, the best we can hope for is a good error message and a quick core dump with no data corruption.

If all users were developers, and only ran their own code, this might be just an irritation. We would get the core dump, track it down, figure out that some idiot had made this change, recode accordingly, and an hour later we'd be good as new. Unfortunately most people run code that they do not understand, and have no way to debug or fix. If a program breaks, all I can do is call support and complain. I am helpless. Users don't care which programmer goofed up. "I bought your product. It doesn't work. I'm taking my business elsewhere!".

If you are going to be delivering binary software to non-developers (that describes 99.999% of all software) you have to trust that what ever platform they run it on will correctly implement all of the interfaces on which your program depends.

The same argument applies, tho not quite as strongly, to independent components in a single system. If components exchange services, and I make an incompatible change to the interfaces of one component, this has the potential to break other components in the same system. I can fix the other components in our system to work with the new changes but:

- this complicates the making of the change.
- we have to ensure that mis-matched component sets are impossible.

## Is every interface graven in stone?

Absolutely not.

You are free to add features that do not change the interface specification (a faster or more robust implementation). In many cases you can add upwards-compatible extensions (all old programs will still work the same way, but new interfaces enable new software to access new functionality).

If the only code that uses my routines is code that I deliver, and I deliver it all in a single package, then I can change my interface at any time.

- customers who have an old package will have the old version of my routine, and all of the code in that package will work with the old routine.
- customers who install the new package will simultaneously get a new version of my routine **and** new versions of all the code that calls it.

But the problem with this is that nobody else is allowed to use my routine.

Another approach would be to send out (to all developers who want to use my routine) an implementation, that they could incorporate into their own products. Later, if I came up with an improved version, I could send that out to all of my developers.

- products written to the old specs would still have an old version of the routine, and simply wouldn't take advantage of my new version.
- newer products, that wanted to take advantage of my new version, could include the new version along with their programs.

It is OK to change interfaces, as long as you can ensure that all clients of the interface will be changed at the same time. It is only when the described module can be delivered independently from the software that uses it that we get into trouble. You simply have to make a choice:

- if you want to be able to change an interface at will, you also have to have control over all of the software that uses the interface.
- if you want to provide an interface to unbundled software (software that can be changed independently) then you must maintain interface stability.

This is a very painful problem, because it would seem to suggest that you have to choose between customer disruption and stifling innovation. Fortunately it is not binary. There is a continuum of stabilities:

- guaranteed for ever
- guaranteed for the life of this product
- guaranteed x months of notice before any incompatible change
- access by special arrangement, which will enable us to manage potentially disruptive changes

- caveat emptor

How stable an interface needs to be depends on how you intend to use it. What is important is that:

1. we be honest about our intentions and set realistic expectations.
2. we have a plan for how we will manage change.

## Interface Stability and Design

So, if we want to expose an interface to unbundled software with high quality requirements, we are not allowed to change it in non-upwards-compatible ways. That sounds like a bother, but if that is the rules, so be it. What has this got to do with architecture and design?

When we design a system, and the interfaces between the independent components, we need to consider all of the different types of change that are likely to happen over the life of this system. When we specify our external component interfaces we need to have high confidence that will be able to accomodate all of the envisioned evolution while preserving those interfaces.

- We might rearrange the distribution of functionality between components to create a simpler (and hence more preservable) interface between them.
- We might design features that we may never implement, just to make sure that the specified interfaces will accomodate them if we ever do decide to build them.
- We might introduce (seemingly) unnatural degrees of abstraction in our services to ensure that they leave us enough slack for future changes.

We must consider which of our interfaces will need to be how stable, and design our system with those stabilities in mind.

# Intro to OS

Sunday, October 23, 2016 2:17 PM

- Operating System (aka. Virtual Machine)
  - Virtualization: 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.
- Interface stability
  - Rewards for standardization to interface
    - Work as a programmer is simplified
    - Program becomes likely portable to any system since all provide the same file services
    - Training time for new programmers is reduced
  - A software interface specification is a form of contract
    - The contract describes an interface and the associated functionality
    - Implementation providers agree that their systems will conform to the specification
    - Developers agree to limit their use of the functionality to what is described in the interface specification
  - Making an incompatible change to the interfaces of one computer has the potential to break other components in the same system.
  - Interfaces are not graven in stone!
    - Free to *add* features that do not change interface specification.
    - Add *upwards-compatible* extensions, meaning all old programs will work the same way but new interfaces enable new software to access new functionality
    - Can be changed if all the clients of the interface will receive the changes at the same time
- Systems and Complexity
  - Problems encountered in many kinds of systems
    - Emergent properties (surprises)
      - Properties not evident in the individual components of a system, but show up when combining those components
      - e.g. bridge swaying to the synchronized footsteps of pedestrians

- Propagation of effects
  - "there are no small changes in a large system"
  - e.g. tire size 13 to 15 inches to improve ride comfort leads to enlarging trunk space, moving seats forwards, knee room too small, back of seats made thinner, reduces ride comfort.
- Incommensurate scaling
  - As a system increases in size or speed, not all parts of it follow the same scaling rules so things stop working
  - e.g. the captain of a modern oil supertanker finds that the ship is so massive that when underway at full speed it takes 12 miles to bring it to a straight line stop—but 12 miles is beyond the horizon as viewed from the ship's bridge
- Trade-offs
  - Waterbed effect: pushing down on a problem at one point causes another problem to pop up somewhere else
  - Binary classification: wish to classify a set of things into two categories based on the presences or absence of some property but we lack a direct measure of the property
- System
  - System
    - a set of interconnected components that has an expected behavior observed at the interface with its environment
  - Depending on point of view we choose to ignore or place importance on certain system components or interfaces
    - Passengers, flight attendants are weight to engine; Engine is discomfort in noise to passengers
  - Sub-system: when a system in one context is a component in another
- Coping with complexity
  - Modularity
    - def: Divide and conquer technique by diving the program into K modules, reduces debugging time by a factor of K.
    - Unyielding foundations rule
      - it is easier to change a module than to change the modularity
    - Robustness principle
      - be tolerant of inputs and strict on outputs
      - Robustness - a system should not be sensitive to modest long term shifts in its environment

## ~~to protect, to mitigate, to limit in its environment~~

- Resilience - a system should continue operating correctly in the face of transient adversity
  - Safety margin principle
    - it is important to track and report out-of-tolerance inputs even if the robustness principle would allow them to be interpreted successfully
  - Bad-news diode
    - good news flows rapidly throughout the organization but bad news often gets confined to the part of the organization that discovers it and does not travel upward
- Abstraction
  - separation of interface from internals, of specifications from implementation
- Layering
  - In designing layers, one builds on a set of mechanisms that is already complete (a lower layer) and uses them to create different complete set of mechanisms (upper layer).
- Hierarchy
  - Arrange a small group of modules and assemble them into a stable, self-contained subsystem with a well defined interface. Then assemble a group of subsystems to produce a larger subsystem.
  - Hierarchy constrains interactions by permitting them only among the components of a subsystem.
  - Binding: choice of a specific implementation among many that are available
- Computer Systems vs other systems
  - The complexity of a computer system is not limited by physical laws and only limited by the designer's ability to understand
  - The rate of change of computer system technology is unprecedented.
    - Rapidly improving technology means that brute-force solutions (buy more memory, wait for a faster processor, use a simpler algorithm) are often the right approach in computer systems.
  - When analog components are composed into a system, the noise from individual components accumulates, thus dominating the system at a certain point limiting the number of components to be cascaded
    - Noise is always a limit on the complexity of analog systems.
  - Static discipline

- requiring that the range or analog values that a device accepts as meaning the digital value of one or zero be wider than the range of analog values that the device puts out when it means digital one or zero
    - Version of the robustness principle
- Maximum depth of composition is the number of devices one can string together
  - For digital systems, noise does not accumulate, and does not constrain depth of composition
  - For digital systems, there are no physical limitation that hardware is subject to such as the speed of light, etc.
- Libraries
  - A collection of object modules with a well-defined interface by which the behavior is invoked.
  - Static libraries (early binding)
    - Linking is performed during the creation of an executable or another object file.
    - All of the modules required by a program are sometimes statically linked and copied into the executable file.
  - Shared library
    - Libraries loaded by programs when they start
    - Advantages
      - Can update libraries and still support programs that want to use older, non-backwards-compatible versions of those libraries.
      - Can override specific libraries or even specific functions in a library when executing a particular program
      - Can do all this while programs are running existing libraries
  - Dynamic library
    - Doesn't require code to be copied and instead just places the name of the library in the binary file only to actually link when the program is run.
    - Must load the library into memory and free after using.
- Application Programming Interface
  - A specification make contain information for routines, data structures, object classes, variables, remote calls etc.
  - Makes it easier for developers to use certain technologies in building applications
    - Reduces the cognitive load on a programmer by abstracting underlying implementations
- Application Binary Interface

- **Application Binary Interface**

- Interface between two program modules, one of which is often a library or operating system at the level of machine code.
- Contains details about how functions are called and in which binary format information should be passed from one program component to the next, or to the operating system in the case of a system call
- ABI similarly defines interfaces between program components but does so at the source code level.

# **Part I**

# **Virtualization**



## 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...

## The Abstraction: The Process

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

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

### THE CRUX OF THE PROBLEM: HOW TO PROVIDE THE ILLUSION OF MANY CPUs?

Although there are only a few physical CPUs available, how can the OS provide the illusion of a nearly-endless supply of said CPUs?

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

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

**TIP: USE TIME SHARING (AND SPACE SHARING)**

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

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

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

## 4.1 The Abstraction: A Process

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

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

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

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

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

**TIP: SEPARATE POLICY AND MECHANISM**

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

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

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

## 4.2 Process API

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

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

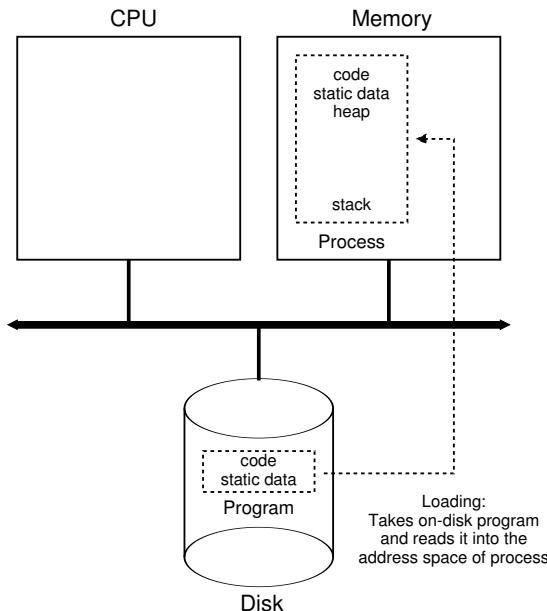


Figure 4.1: Loading: From Program To Process

### 4.3 Process Creation: A Little More Detail

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

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

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

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

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

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

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

## 4.4 Process States

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

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

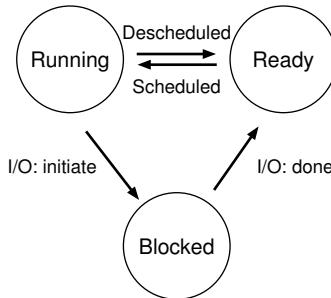


Figure 4.2: Process: State Transitions

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

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

Time	Process <sub>0</sub>	Process <sub>1</sub>	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process <sub>0</sub> now done
5	-	Running	
6	-	Running	
7	-	Running	
8	-	Running	Process <sub>1</sub> now done

Figure 4.3: Tracing Process State: CPU Only

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

More specifically, Process<sub>0</sub> initiates an I/O and becomes blocked waiting for it to complete; processes become blocked, for example, when read-

Time	Process <sub>0</sub>	Process <sub>1</sub>	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process <sub>0</sub> initiates I/O
4	Blocked	Running	Process <sub>0</sub> is blocked,
5	Blocked	Running	so Process <sub>1</sub> runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process <sub>1</sub> now done
9	Running	—	
10	Running	—	Process <sub>0</sub> now done

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

ing from a disk or waiting for a packet from a network. The OS recognizes Process<sub>0</sub> is not using the CPU and starts running Process<sub>1</sub>. While Process<sub>1</sub> is running, the I/O completes, moving Process<sub>0</sub> back to ready. Finally, Process<sub>1</sub> finishes, and Process<sub>0</sub> 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<sub>1</sub> while Process<sub>0</sub> issued an I/O; doing so improves resource utilization by keeping the CPU busy. Second, the system decided not to switch back to Process<sub>0</sub> when its I/O completed; it is not clear if this is a good decision or not. What do you think? These types of decisions are made by the OS **scheduler**, a topic we will discuss a few chapters in the future.

## 4.5 Data Structures

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

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

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

```

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

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

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

```

Figure 4.5: The xv6 Proc Structure

You can also see from the figure that there are some other states a process can be in, beyond running, ready, and blocked. Sometimes a system will have an **initial** state that the process is in when it is being created. Also, a process could be placed in a **final** state where it has exited but has not yet been cleaned up (in UNIX-based systems, this is called the **zombie** state<sup>1</sup>). 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.

---

<sup>1</sup>Yes, the zombie state. Just like real zombies, these zombies are relatively easy to kill. However, different techniques are usually recommended.

**ASIDE: DATA STRUCTURE — THE PROCESS LIST**

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

## 4.6 Summary

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

## References

- [BH70] "The Nucleus of a Multiprogramming System"  
Per Brinch Hansen  
Communications of the ACM, Volume 13, Number 4, April 1970  
*This paper introduces one of the first **microkernels** in operating systems history, called Nucleus. The idea of smaller, more minimal systems is a theme that rears its head repeatedly in OS history; it all began with Brinch Hansen's work described herein.*
- [CK+08] "The xv6 Operating System"  
Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich  
From: <http://pdos.csail.mit.edu/6.828/2008/index.html>  
*The coolest real and little OS in the world. Download and play with it to learn more about the details of how operating systems actually work.*
- [DV66] "Programming Semantics for Multiprogrammed Computations"  
Jack B. Dennis and Earl C. Van Horn  
Communications of the ACM, Volume 9, Number 3, March 1966  
*This paper defined many of the early terms and concepts around building multiprogrammed systems.*
- [L+75] "Policy/mechanism separation in Hydra"  
R. Levin, E. Cohen, W. Corwin, F. Pollack, W. Wulf  
SOSP 1975  
*An early paper about how to structure operating systems in a research OS known as Hydra. While Hydra never became a mainstream OS, some of its ideas influenced OS designers.*
- [V+65] "Structure of the Multics Supervisor"  
V.A. Vyssotsky, F. J. Corbato, R. M. Graham  
Fall Joint Computer Conference, 1965  
*An early paper on Multics, which described many of the basic ideas and terms that we find in modern systems. Some of the vision behind computing as a utility are finally being realized in modern cloud systems.*

## Homework

### ASIDE: SIMULATION HOMEWORKS

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

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

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

## Questions

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

- cess that issued it is not necessarily run right away; rather, whatever was running at the time keeps running. What happens when you run this combination of processes? (`./process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -I IO_RUN_LATER -c -p`) Are system resources being effectively utilized?
7. Now run the same processes, but with `-I IO_RUN_IMMEDIATE` set, which immediately runs the process that issued the I/O. How does this behavior differ? Why might running a process that just completed an I/O again be a good idea?
  8. Now run with some randomly generated processes, e.g., `-s 1 -l 3:50,3:50, -s 2 -l 3:50,3:50, -s 3 -l 3:50,3:50`. See if you can predict how the trace will turn out. What happens when you use `-I IO_RUN_IMMEDIATE` vs. `-I IO_RUN_LATER`? What happens when you use `-S SWITCH_ON_IO` vs. `-S SWITCH_ON_END`?

## 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 ease of use as well as utility?

### 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<sup>1</sup>. 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!

<sup>1</sup>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.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 int
6 main(int argc, char *argv[])
7 {
8     printf("hello world (pid:%d)\n", (int) getpid());
9     int rc = fork();
10    if (rc < 0) {           // fork failed; exit
11        fprintf(stderr, "fork failed\n");
12        exit(1);
13    } else if (rc == 0) {   // child (new process)
14        printf("hello, I am child (pid:%d)\n", (int) getpid());
15    } else {                // parent goes down this path (main)
16        printf("hello, I am parent of %d (pid:%d)\n",
17               rc, (int) getpid());
18    }
19    return 0;
20 }
```

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.

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).

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5
6 int
7 main(int argc, char *argv[])
8 {
9     printf("hello world (pid:%d)\n", (int) getpid());
10    int rc = fork();
11    if (rc < 0) {           // fork failed; exit
12        fprintf(stderr, "fork failed\n");
13        exit(1);
14    } else if (rc == 0) {   // child (new process)
15        printf("hello, I am child (pid:%d)\n", (int) getpid());
16    } else {                // parent goes down this path (main)
17        int wc = wait(NULL);
18        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
19               rc, wc, (int) getpid());
20    }
21    return 0;
22 }
```

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

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 (wc: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<sup>2</sup>. 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<sup>3</sup>. This system call is useful when you want to run a program that is different from the calling program. For example, calling `fork()` 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, page 5).

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 (wc:29384) (pid:29383)
prompt>
```

---

<sup>2</sup>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."

<sup>3</sup>Actually, there are six variants of `exec()`: `exec1()`, `execle()`, `execlp()`, `execv()`, and `execvp()`. Read the man pages to learn more.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <sys/wait.h>
6
7 int
8 main(int argc, char *argv[])
9 {
10     printf("hello world (pid:%d)\n", (int) getpid());
11     int rc = fork();
12     if (rc < 0) {           // fork failed; exit
13         fprintf(stderr, "fork failed\n");
14         exit(1);
15     } else if (rc == 0) {   // child (new process)
16         printf("hello, I am child (pid:%d)\n", (int) getpid());
17         char *myargs[3];
18         myargs[0] = strdup("wc");    // program: "wc" (word count)
19         myargs[1] = strdup("p3.c"); // argument: file to count
20         myargs[2] = NULL;          // marks end of array
21         execvp(myargs[0], myargs); // runs word count
22         printf("this shouldn't print out");
23     } else {               // parent goes down this path (main)
24         int wc = wait(NULL);
25         printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
26                rc, wc, (int) getpid());
27     }
28     return 0;
29 }
```

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

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 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.

**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.

The shell is just a user program<sup>4</sup>. 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
```

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 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>
```

---

<sup>4</sup>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.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <fcntl.h>
6 #include <sys/wait.h>
7
8 int
9 main(int argc, char *argv[])
10 {
11     int rc = fork();
12     if (rc < 0) {           // fork failed; exit
13         fprintf(stderr, "fork failed\n");
14         exit(1);
15     } else if (rc == 0) {   // child: redirect standard output to a file
16         close(STDOUT_FILENO);
17         open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
18
19         // now exec "wc"...
20         char *myargs[3];
21         myargs[0] = strdup("wc");    // program: "wc" (word count)
22         myargs[1] = strdup("p4.c");  // argument: file to count
23         myargs[2] = NULL;          // marks end of array
24         execvp(myargs[0], myargs); // runs word count
25     } else {                // parent goes down this path (main)
26         int wc = wait(NULL);
27     }
28     return 0;
29 }
```

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

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.

**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...

## 5.5 Other Parts Of The API

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 go to sleep, die, and other useful imperatives. In fact, the entire signals subsystem provides a rich infrastructure to deliver external events to processes, including ways to receive and process those signals.

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. 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.

## 5.6 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.

## References

[C63] "A Multiprocessor System Design"

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"

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.*

[L83] "Hints for Computer Systems Design"

Butler Lampson

ACM Operating Systems Review, 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.*

[SR05] "Advanced Programming in the UNIX Environment"

W. Richard Stevens and 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.*

**ASIDE: CODING HOMEWORKS**

Coding homeworks are small exercises where you write code to run on a real machine to get some experience with some of the basic APIs that modern operating systems have to offer. After all, you are (probably) a computer scientist, and therefore should like to code, right? 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<sup>5</sup>, 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 `exec1()`, `execle()`, `execlp()`, `execv()`, `execvp()`, and `execvp()`. 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?

---

<sup>5</sup>If you don't like to code, but want to become a computer scientist, this means you need to either (a) become really good at the theory of computer science, or (b) perhaps rethink this whole "computer science" thing you've been telling everyone about.

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.

## 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 <b>return</b> from main
Free memory of process	
Remove from process list	

Figure 6.1: Direct Execution Protocol (Without Limits)

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

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

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

## 6.2 Problem #1: Restricted Operations

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

### THE CRUX: HOW TO PERFORM RESTRICTED OPERATIONS

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

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

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

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

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

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

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

**TIP: USE PROTECTED CONTROL TRANSFER**

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

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

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

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

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

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

---

<sup>1</sup>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 args Fill kernel stack with reg/PC <b>return-from-trap</b>	restore regs from kernel stack move to user mode jump to main	Run main() ...
		Call system call <b>trap</b> into OS
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall <b>return-from-trap</b>	restore regs from kernel stack move to user mode jump to PC after trap	...
		return from main <b>trap</b> (via <code>exit()</code> )
Free memory of process Remove from process list		

Figure 6.2: Limited Direct Execution Protocol

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

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

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

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

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

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

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

### 6.3 Problem #2: Switching Between Processes

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

#### THE CRUX: HOW TO REGAIN CONTROL OF THE CPU

How can the operating system **regain control** of the CPU so that it can switch between processes?

#### A Cooperative Approach: Wait For System Calls

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

**TIP: DEALING WITH APPLICATION MISBEHAVIOR**

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

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

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

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

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

### A Non-Cooperative Approach: The OS Takes Control

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

**THE CRUX: HOW TO GAIN CONTROL WITHOUT COOPERATION**

How can the OS gain control of the CPU even if processes are not being cooperative? What can the OS do to ensure a rogue process does not take over the machine?

**TIP: USE THE TIMER INTERRUPT TO REGAIN CONTROL**

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

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

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

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

## Saving and Restoring Context

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

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

OS @ boot (kernel mode)	Hardware	
initialize trap table	remember addresses of... syscall handler timer handler	
start interrupt timer	start timer interrupt CPU in X ms	
OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call <code>switch()</code> routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) <b>return-from-trap (into B)</b>	restore regs(B) from k-stack(B) move to user mode jump to B's PC	Process B
		...
		...

Figure 6.3: Limited Direct Execution Protocol (Timer Interrupt)

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

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

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

```

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

```

Figure 6.4: The xv6 Context Switch Code

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

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

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

## 6.4 Worried About Concurrency?

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

**ASIDE: HOW LONG CONTEXT SWITCHES TAKE**

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

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

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

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

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

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

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

## 6.5 Summary

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

**TIP: REBOOT IS USEFUL**

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

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

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

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

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

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

## References

- [A79] "Alto User's Handbook"  
 Xerox Palo Alto Research Center, September 1979  
 Available: <http://history-computer.com/Library/AltoUsersHandbook.pdf>  
*An amazing system, way ahead of its time. Became famous because Steve Jobs visited, took notes, and built Lisa and eventually Mac.*
- [C+04] "Microreboot — A Technique for Cheap Recovery"  
 George Canea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, Armando Fox  
 OSDI '04, San Francisco, CA, December 2004  
*An excellent paper pointing out how far one can go with reboot in building more robust systems.*
- [I11] "Intel 64 and IA-32 Architectures Software Developer's Manual"  
 Volume 3A and 3B: System Programming Guide  
 Intel Corporation, January 2011
- [K+61] "One-Level Storage System"  
 T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner  
 IRE Transactions on Electronic Computers, April 1962  
*The Atlas pioneered much of what you see in modern systems. However, this paper is not the best one to read. If you were to only read one, you might try the historical perspective below [L78].*
- [L78] "The Manchester Mark I and Atlas: A Historical Perspective"  
 S. H. Lavington  
 Communications of the ACM, 21:1, January 1978  
*A history of the early development of computers and the pioneering efforts of Atlas.*
- [M+63] "A Time-Sharing Debugging System for a Small Computer"  
 J. McCarthy, S. Boilen, E. Fredkin, J. C. R. Licklider  
 AFIPS '63 (Spring), May, 1963, New York, USA  
*An early paper about time-sharing that refers to using a timer interrupt; the quote that discusses it: "The basic task of the channel 17 clock routine is to decide whether to remove the current user from core and if so to decide which user program to swap in as he goes out."*
- [MS96] "lmbench: Portable tools for performance analysis"  
 Larry McVoy and Carl Staelin  
 USENIX Annual Technical Conference, January 1996  
*A fun paper about how to measure a number of different things about your OS and its performance. Download lmbench and give it a try.*
- [M11] "Mac OS 9"  
 January 2011  
 Available: [http://en.wikipedia.org/wiki/Mac\\_OS\\_9](http://en.wikipedia.org/wiki/Mac_OS_9)
- [O90] "Why Aren't Operating Systems Getting Faster as Fast as Hardware?"  
 J. Ousterhout  
 USENIX Summer Conference, June 1990  
*A classic paper on the nature of operating system performance.*
- [P10] "The Single UNIX Specification, Version 3"  
 The Open Group, May 2010  
 Available: <http://www.unix.org/version3/>  
*This is hard and painful to read, so probably avoid it if you can.*
- [S07] "The Geometry of Innocent Flesh on the Bone:  
 Return-into-libc without Function Calls (on the x86)"  
 Hovav Shacham  
 CCS '07, October 2007  
*One of those awesome, mind-blowing ideas that you'll see in research from time to time. The author shows that if you can jump into code arbitrarily, you can essentially stitch together any code sequence you like (given a large code base); read the paper for the details. The technique makes it even harder to defend against malicious attacks, alas.*

## Homework (Measurement)

### ASIDE: MEASUREMENT HOMEWORKS

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

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

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

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

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

## 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 ease of use as well as utility?

### 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<sup>1</sup>. 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!

<sup>1</sup>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.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 int
6 main(int argc, char *argv[])
7 {
8     printf("hello world (pid:%d)\n", (int) getpid());
9     int rc = fork();
10    if (rc < 0) {           // fork failed; exit
11        fprintf(stderr, "fork failed\n");
12        exit(1);
13    } else if (rc == 0) {   // child (new process)
14        printf("hello, I am child (pid:%d)\n", (int) getpid());
15    } else {                // parent goes down this path (main)
16        printf("hello, I am parent of %d (pid:%d)\n",
17               rc, (int) getpid());
18    }
19    return 0;
20 }
```

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.

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).

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5
6 int
7 main(int argc, char *argv[])
8 {
9     printf("hello world (pid:%d)\n", (int) getpid());
10    int rc = fork();
11    if (rc < 0) {           // fork failed; exit
12        fprintf(stderr, "fork failed\n");
13        exit(1);
14    } else if (rc == 0) {   // child (new process)
15        printf("hello, I am child (pid:%d)\n", (int) getpid());
16    } else {                // parent goes down this path (main)
17        int wc = wait(NULL);
18        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
19               rc, wc, (int) getpid());
20    }
21    return 0;
22 }
```

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

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 (wc: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<sup>2</sup>. 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<sup>3</sup>. This system call is useful when you want to run a program that is different from the calling program. For example, calling `fork()` 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, page 5).

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 (wc:29384) (pid:29383)
prompt>
```

---

<sup>2</sup>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."

<sup>3</sup>Actually, there are six variants of `exec()`: `exec1()`, `execle()`, `execlp()`, `execv()`, and `execvp()`. Read the man pages to learn more.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <sys/wait.h>
6
7 int
8 main(int argc, char *argv[])
9 {
10     printf("hello world (pid:%d)\n", (int) getpid());
11     int rc = fork();
12     if (rc < 0) {           // fork failed; exit
13         fprintf(stderr, "fork failed\n");
14         exit(1);
15     } else if (rc == 0) {   // child (new process)
16         printf("hello, I am child (pid:%d)\n", (int) getpid());
17         char *myargs[3];
18         myargs[0] = strdup("wc");    // program: "wc" (word count)
19         myargs[1] = strdup("p3.c"); // argument: file to count
20         myargs[2] = NULL;          // marks end of array
21         execvp(myargs[0], myargs); // runs word count
22         printf("this shouldn't print out");
23     } else {               // parent goes down this path (main)
24         int wc = wait(NULL);
25         printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
26                rc, wc, (int) getpid());
27     }
28     return 0;
29 }
```

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

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 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.

**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.

The shell is just a user program<sup>4</sup>. 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
```

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 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>
```

---

<sup>4</sup>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.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <fcntl.h>
6 #include <sys/wait.h>
7
8 int
9 main(int argc, char *argv[])
10 {
11     int rc = fork();
12     if (rc < 0) {           // fork failed; exit
13         fprintf(stderr, "fork failed\n");
14         exit(1);
15     } else if (rc == 0) {   // child: redirect standard output to a file
16         close(STDOUT_FILENO);
17         open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
18
19         // now exec "wc"...
20         char *myargs[3];
21         myargs[0] = strdup("wc");    // program: "wc" (word count)
22         myargs[1] = strdup("p4.c"); // argument: file to count
23         myargs[2] = NULL;          // marks end of array
24         execvp(myargs[0], myargs); // runs word count
25     } else {                // parent goes down this path (main)
26         int wc = wait(NULL);
27     }
28     return 0;
29 }
```

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

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.

**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...

## 5.5 Other Parts Of The API

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 go to sleep, die, and other useful imperatives. In fact, the entire signals subsystem provides a rich infrastructure to deliver external events to processes, including ways to receive and process those signals.

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. 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.

## 5.6 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.

## References

[C63] "A Multiprocessor System Design"

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"

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.*

[L83] "Hints for Computer Systems Design"

Butler Lampson

ACM Operating Systems Review, 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.*

[SR05] "Advanced Programming in the UNIX Environment"

W. Richard Stevens and 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.*

**ASIDE: CODING HOMEWORKS**

Coding homeworks are small exercises where you write code to run on a real machine to get some experience with some of the basic APIs that modern operating systems have to offer. After all, you are (probably) a computer scientist, and therefore should like to code, right? 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<sup>5</sup>, 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 `exec1()`, `execle()`, `execlp()`, `execv()`, `execvp()`, and `execvp()`. 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?

---

<sup>5</sup>If you don't like to code, but want to become a computer scientist, this means you need to either (a) become really good at the theory of computer science, or (b) perhaps rethink this whole "computer science" thing you've been telling everyone about.

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.

## 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 <b>return</b> from main
Free memory of process	
Remove from process list	

Figure 6.1: Direct Execution Protocol (Without Limits)

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

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

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

## 6.2 Problem #1: Restricted Operations

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

### THE CRUX: HOW TO PERFORM RESTRICTED OPERATIONS

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

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

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

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

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

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

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

**TIP: USE PROTECTED CONTROL TRANSFER**

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

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

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

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

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

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

---

<sup>1</sup>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 args Fill kernel stack with reg/PC <b>return-from-trap</b>	restore regs from kernel stack move to user mode jump to main	Run main() ...
		Call system call <b>trap</b> into OS
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall <b>return-from-trap</b>	restore regs from kernel stack move to user mode jump to PC after trap	...
		return from main <b>trap</b> (via <code>exit()</code> )
Free memory of process Remove from process list		

Figure 6.2: Limited Direct Execution Protocol

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

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

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

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

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

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

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

### 6.3 Problem #2: Switching Between Processes

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

#### THE CRUX: HOW TO REGAIN CONTROL OF THE CPU

How can the operating system **regain control** of the CPU so that it can switch between processes?

#### A Cooperative Approach: Wait For System Calls

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

**TIP: DEALING WITH APPLICATION MISBEHAVIOR**

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

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

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

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

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

### A Non-Cooperative Approach: The OS Takes Control

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

**THE CRUX: HOW TO GAIN CONTROL WITHOUT COOPERATION**

How can the OS gain control of the CPU even if processes are not being cooperative? What can the OS do to ensure a rogue process does not take over the machine?

**TIP: USE THE TIMER INTERRUPT TO REGAIN CONTROL**

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

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

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

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

## Saving and Restoring Context

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

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

OS @ boot (kernel mode)	Hardware	
initialize trap table	remember addresses of... syscall handler timer handler	
start interrupt timer	start timer interrupt CPU in X ms	
OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call <code>switch()</code> routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) <b>return-from-trap (into B)</b>	restore regs(B) from k-stack(B) move to user mode jump to B's PC	Process B
		...
		...

Figure 6.3: Limited Direct Execution Protocol (Timer Interrupt)

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

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

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

```

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

```

Figure 6.4: The xv6 Context Switch Code

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

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

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

## 6.4 Worried About Concurrency?

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

**ASIDE: HOW LONG CONTEXT SWITCHES TAKE**

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

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

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

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

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

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

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

## 6.5 Summary

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

**TIP: REBOOT IS USEFUL**

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

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

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

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

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

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

## References

- [A79] "Alto User's Handbook"  
 Xerox Palo Alto Research Center, September 1979  
 Available: <http://history-computer.com/Library/AltoUsersHandbook.pdf>  
*An amazing system, way ahead of its time. Became famous because Steve Jobs visited, took notes, and built Lisa and eventually Mac.*
- [C+04] "Microreboot — A Technique for Cheap Recovery"  
 George Canea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, Armando Fox  
 OSDI '04, San Francisco, CA, December 2004  
*An excellent paper pointing out how far one can go with reboot in building more robust systems.*
- [I11] "Intel 64 and IA-32 Architectures Software Developer's Manual"  
 Volume 3A and 3B: System Programming Guide  
 Intel Corporation, January 2011
- [K+61] "One-Level Storage System"  
 T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner  
 IRE Transactions on Electronic Computers, April 1962  
*The Atlas pioneered much of what you see in modern systems. However, this paper is not the best one to read. If you were to only read one, you might try the historical perspective below [L78].*
- [L78] "The Manchester Mark I and Atlas: A Historical Perspective"  
 S. H. Lavington  
 Communications of the ACM, 21:1, January 1978  
*A history of the early development of computers and the pioneering efforts of Atlas.*
- [M+63] "A Time-Sharing Debugging System for a Small Computer"  
 J. McCarthy, S. Boilen, E. Fredkin, J. C. R. Licklider  
 AFIPS '63 (Spring), May, 1963, New York, USA  
*An early paper about time-sharing that refers to using a timer interrupt; the quote that discusses it: "The basic task of the channel 17 clock routine is to decide whether to remove the current user from core and if so to decide which user program to swap in as he goes out."*
- [MS96] "lmbench: Portable tools for performance analysis"  
 Larry McVoy and Carl Staelin  
 USENIX Annual Technical Conference, January 1996  
*A fun paper about how to measure a number of different things about your OS and its performance. Download lmbench and give it a try.*
- [M11] "Mac OS 9"  
 January 2011  
 Available: [http://en.wikipedia.org/wiki/Mac\\_OS\\_9](http://en.wikipedia.org/wiki/Mac_OS_9)
- [O90] "Why Aren't Operating Systems Getting Faster as Fast as Hardware?"  
 J. Ousterhout  
 USENIX Summer Conference, June 1990  
*A classic paper on the nature of operating system performance.*
- [P10] "The Single UNIX Specification, Version 3"  
 The Open Group, May 2010  
 Available: <http://www.unix.org/version3/>  
*This is hard and painful to read, so probably avoid it if you can.*
- [S07] "The Geometry of Innocent Flesh on the Bone:  
 Return-into-libc without Function Calls (on the x86)"  
 Hovav Shacham  
 CCS '07, October 2007  
*One of those awesome, mind-blowing ideas that you'll see in research from time to time. The author shows that if you can jump into code arbitrarily, you can essentially stitch together any code sequence you like (given a large code base); read the paper for the details. The technique makes it even harder to defend against malicious attacks, alas.*

## Homework (Measurement)

### ASIDE: MEASUREMENT HOMEWORKS

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

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

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

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

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

**KILL(2)****Linux Programmer's Manual****KILL(2)****NAME**[top](#)

kill - send signal to a process

**SYNOPSIS**[top](#)

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

Feature Test Macro Requirements for glibc (see [feature\\_test\\_macros\(7\)](#)):

```
kill(): _POSIX_C_SOURCE
```

**DESCRIPTION**[top](#)

The **kill()** system call can be used to send any signal to any process group or process.

If *pid* is positive, then signal *sig* is sent to the process with the ID specified by *pid*.

If *pid* equals 0, then *sig* is sent to every process in the process group of the calling process.

If *pid* equals -1, then *sig* is sent to every process for which the calling process has permission to send signals, except for process 1 (*init*), but see below.

If *pid* is less than -1, then *sig* is sent to every process in the process group whose ID is *-pid*.

If *sig* is 0, then no signal is sent, but existence and permission checks are still performed; this can be used to check for the existence of a process ID or process group ID that the caller is permitted to signal.

For a process to have permission to send a signal, it must either be privileged (under Linux: have the **CAP\_KILL** capability in the user namespace of the target process), or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the target process. In the case of **SIGCONT**, it suffices when the sending

and receiving processes belong to the same session. (Historically, the rules were different; see NOTES.)

## RETURN VALUE

[top](#)

On success (at least one signal was sent), zero is returned. On error, -1 is returned, and *errno* is set appropriately.

## ERRORS

[top](#)

**EINVAL** An invalid signal was specified.

**EPERM** The process does not have permission to send the signal to any of the target processes.

**ESRCH** The process or process group does not exist. Note that an existing process might be a zombie, a process that has terminated execution, but has not yet been *wait(2)*ed for.

## CONFORMING TO

[top](#)

POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD.

## NOTES

[top](#)

The only signals that can be sent to process ID 1, the *init* process, are those for which *init* has explicitly installed signal handlers. This is done to assure the system is not brought down accidentally.

POSIX.1 requires that *kill(-1,sig)* send *sig* to all processes that the calling process may send signals to, except possibly for some implementation-defined system processes. Linux allows a process to signal itself, but on Linux the call *kill(-1,sig)* does not signal the calling process.

POSIX.1 requires that if a process sends a signal to itself, and the sending thread does not have the signal blocked, and no other thread has it unblocked or is waiting for it in *sigwait(3)*, at least one unblocked signal must be delivered to the sending thread before the *kill()* returns.

### Linux notes

Across different kernel versions, Linux has enforced different rules for the permissions required for an unprivileged process to send a signal to another process. In kernels 1.0 to 1.2.2, a signal could be sent if the effective user ID of the sender matched effective user ID of the target, or the real user ID of the sender matched the real user ID of the target. From kernel 1.2.3 until 1.3.77, a signal could be sent if the effective user ID of the sender matched either

the real or effective user ID of the target. The current rules, which conform to POSIX.1, were adopted in kernel 1.3.78.

## BUGS

[top](#)

In 2.6 kernels up to and including 2.6.7, there was a bug that meant that when sending signals to a process group, `kill()` failed with the error `EPERM` if the caller did not have permission to send the signal to *any* (rather than *all*) of the members of the process group. Notwithstanding this error return, the signal was still delivered to all of the processes for which the caller had permission to signal.

## SEE ALSO

[top](#)

[`\_exit\(2\)`](#), [`killpg\(2\)`](#), [`signal\(2\)`](#), [`tkill\(2\)`](#), [`exit\(3\)`](#), [`sigqueue\(3\)`](#), [`capabilities\(7\)`](#), [`credentials\(7\)`](#), [`signal\(7\)`](#)

## COLOPHON

[top](#)

This page is part of release 4.08 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at  
<https://www.kernel.org/doc/man-pages/>.

Linux

2016-07-17

KILL(2)

---

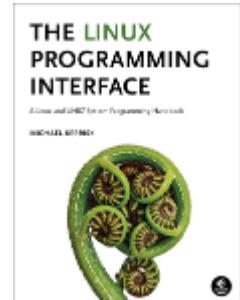
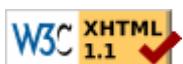
## Copyright and license for this manual page

---

HTML rendering created 2016-10-08 by Michael Kerrisk, author of *The Linux Programming Interface*, maintainer of the Linux *man-pages* project.

For details of in-depth Linux/UNIX system programming training courses that I teach, look [here](#).

Hosting by [jambit GmbH](#).



**SIGNAL(2)****Linux Programmer's Manual****SIGNAL(2)****NAME**[top](#)

signal - ANSI C signal handling

**SYNOPSIS**[top](#)

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

**DESCRIPTION**[top](#)

The behavior of `signal()` varies across UNIX versions, and has also varied historically across different versions of Linux. **Avoid its use:** use `sigaction(2)` instead. See *Portability* below.

`signal()` sets the disposition of the signal `signum` to `handler`, which is either `SIG_IGN`, `SIG_DFL`, or the address of a programmer-defined function (a "signal handler").

If the signal `signum` is delivered to the process, then one of the following happens:

- \* If the disposition is set to `SIG_IGN`, then the signal is ignored.
- \* If the disposition is set to `SIG_DFL`, then the default action associated with the signal (see `signal(7)`) occurs.
- \* If the disposition is set to a function, then first either the disposition is reset to `SIG_DFL`, or the signal is blocked (see *Portability* below), and then `handler` is called with argument `signum`. If invocation of the handler caused the signal to be blocked, then the signal is unblocked upon return from the handler.

The signals `SIGKILL` and `SIGSTOP` cannot be caught or ignored.

**RETURN VALUE**[top](#)

`signal()` returns the previous value of the signal handler, or **SIG\_ERR** on error. In the event of an error, `errno` is set to indicate the cause.

## ERRORS

[top](#)

**EINVAL** `signum` is invalid.

## CONFORMING TO

[top](#)

POSIX.1-2001, POSIX.1-2008, C89, C99.

## NOTES

[top](#)

The effects of `signal()` in a multithreaded process are unspecified.

According to POSIX, the behavior of a process is undefined after it ignores a **SIGFPE**, **SIGILL**, or **SIGSEGV** signal that was not generated by `kill(2)` or `raise(3)`. Integer division by zero has undefined result. On some architectures it will generate a **SIGFPE** signal. (Also dividing the most negative integer by -1 may generate **SIGFPE**.) Ignoring this signal might lead to an endless loop.

See `sigaction(2)` for details on what happens when **SIGCHLD** is set to **SIG\_IGN**.

See `signal(7)` for a list of the async-signal-safe functions that can be safely called from inside a signal handler.

The use of `sighandler_t` is a GNU extension, exposed if **\_GNU\_SOURCE** is defined; glibc also defines (the BSD-derived) `sig_t` if **\_BSD\_SOURCE** (glibc 2.19 and earlier) or **\_DEFAULT\_SOURCE** (glibc 2.19 and later) is defined. Without use of such a type, the declaration of `signal()` is the somewhat harder to read:

```
void ( *signal(int signum, void (*handler)(int)) ) (int);
```

## Portability

The only portable use of `signal()` is to set a signal's disposition to **SIG\_DFL** or **SIG\_IGN**. The semantics when using `signal()` to establish a signal handler vary across systems (and POSIX.1 explicitly permits this variation); **do not use it for this purpose**.

POSIX.1 solved the portability mess by specifying `sigaction(2)`, which provides explicit control of the semantics when a signal handler is invoked; use that interface instead of `signal()`.

In the original UNIX systems, when a handler that was established using `signal()` was invoked by the delivery of a signal, the disposition of the signal would be reset to **SIG\_DFL**, and the system did not block delivery of further instances of the signal. This is

equivalent to calling `sigaction(2)` with the following flags:

```
sa.sa_flags = SA_RESETHAND | SA_NODEFER;
```

System V also provides these semantics for `signal()`. This was bad because the signal might be delivered again before the handler had a chance to reestablish itself. Furthermore, rapid deliveries of the same signal could result in recursive invocations of the handler.

BSD improved on this situation, but unfortunately also changed the semantics of the existing `signal()` interface while doing so. On BSD, when a signal handler is invoked, the signal disposition is not reset, and further instances of the signal are blocked from being delivered while the handler is executing. Furthermore, certain blocking system calls are automatically restarted if interrupted by a signal handler (see `signal(7)`). The BSD semantics are equivalent to calling `sigaction(2)` with the following flags:

```
sa.sa_flags = SA_RESTART;
```

The situation on Linux is as follows:

- \* The kernel's `signal()` system call provides System V semantics.
- \* By default, in glibc 2 and later, the `signal()` wrapper function does not invoke the kernel system call. Instead, it calls `sigaction(2)` using flags that supply BSD semantics. This default behavior is provided as long as a suitable feature test macro is defined: `_BSD_SOURCE` on glibc 2.19 and earlier or `_DEFAULT_SOURCE` in glibc 2.19 and later. (By default, these macros are defined; see `feature_test_macros(7)` for details.) If such a feature test macro is not defined, then `signal()` provides System V semantics.

## SEE ALSO

[top](#)

`kill(1)`, `alarm(2)`, `kill(2)`, `killpg(2)`, `pause(2)`, `sigaction(2)`, `signalfd(2)`, `sigpending(2)`, `sigprocmask(2)`, `sigsuspend(2)`, `bsd_signal(3)`, `raise(3)`, `siginterrupt(3)`, `sigqueue(3)`, `sigsetops(3)`, `sigvec(3)`, `sysv_signal(3)`, `signal(7)`

## COLOPHON

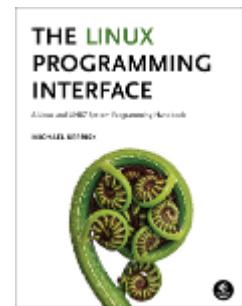
[top](#)

This page is part of release 4.08 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at  
<https://www.kernel.org/doc/man-pages/>.

HTML rendering created 2016-10-08 by Michael Kerrisk, author of *The Linux Programming Interface*, maintainer of the [Linux man-pages project](#).

For details of in-depth **Linux/UNIX system programming training courses** that I teach, look [here](#).

Hosting by [jambit GmbH](#).



## 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**<sup>1</sup>.

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:

$$T_{turnaround} = T_{completion} - T_{arrival} \quad (7.1)$$

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)**.

---

<sup>1</sup>Said in the same way you would say "A fully-operational Death Star."

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

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

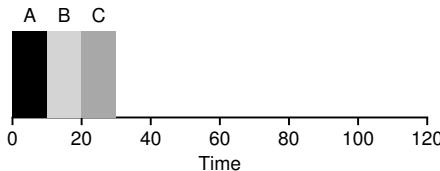


Figure 7.1: FIFO Simple Example

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

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

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

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

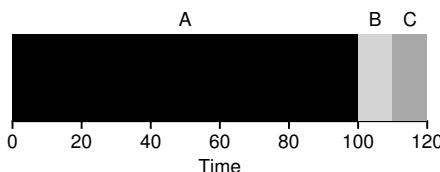


Figure 7.2: Why FIFO Is Not That Great

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

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

## TIP: THE PRINCIPLE OF SJF

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

you see the person in front of you with three carts full of provisions and a checkbook out; it’s going to be a while<sup>2</sup>.

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

## 7.4 Shortest Job First (SJF)

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

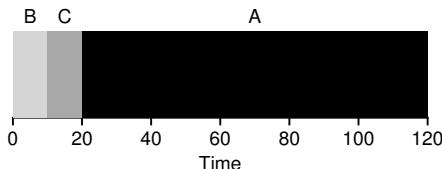


Figure 7.3: SJF Simple Example

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

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

---

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

**ASIDE: PREEMPTIVE SCHEDULERS**

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

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

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

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

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

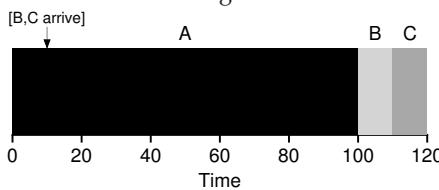


Figure 7.4: SJF With Late Arrivals From B and C

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

## 7.5 Shortest Time-to-Completion First (STCF)

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

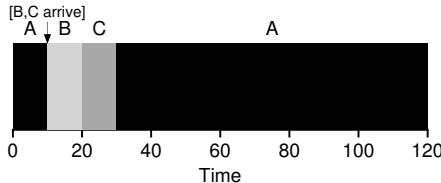


Figure 7.5: STCF Simple Example

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

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

## 7.6 A New Metric: Response Time

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

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

$$T_{\text{response}} = T_{\text{firstrun}} - T_{\text{arrival}} \quad (7.2)$$

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

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

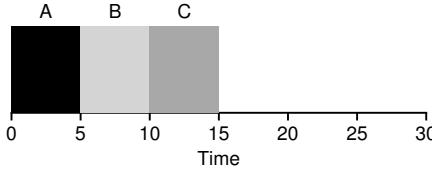


Figure 7.6: SJF Again (Bad for Response Time)

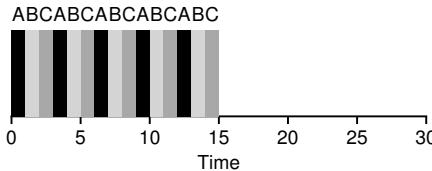


Figure 7.7: Round Robin (Good for Response Time)

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

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

## 7.7 Round Robin

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

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

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

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

**TIP: AMORTIZATION CAN REDUCE COSTS**

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

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

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

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

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

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.

---

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

**TIP: OVERLAP ENABLES HIGHER UTILIZATION**

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

## 7.8 Incorporating I/O

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

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

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

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

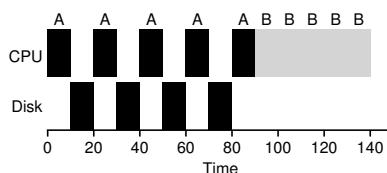


Figure 7.8: Poor Use of Resources

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

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

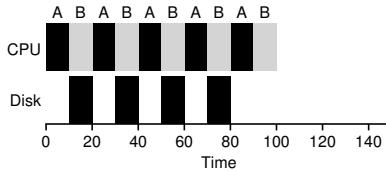


Figure 7.9: Overlap Allows Better Use of Resources

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

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

## 7.9 No More Oracle

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

## 7.10 Summary

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

## References

- [B+79] "The Convoy Phenomenon"  
 M. Blasgen, J. Gray, M. Mitoma, T. Price  
*ACM Operating Systems Review, 13:2, April 1979*  
*Perhaps the first reference to convoys, which occurs in databases as well as the OS.*
- [C54] "Priority Assignment in Waiting Line Problems"  
 A. Cobham  
*Journal of Operations Research, 2:70, pages 70–76, 1954*  
*The pioneering paper on using an SJF approach in scheduling the repair of machines.*
- [K64] "Analysis of a Time-Shared Processor"  
 Leonard Kleinrock  
*Naval Research Logistics Quarterly, 11:1, pages 59–73, March 1964*  
*May be the first reference to the round-robin scheduling algorithm; certainly one of the first analyses of said approach to scheduling a time-shared system.*
- [CK68] "Computer Scheduling Methods and their Countermeasures"  
 Edward G. Coffman and Leonard Kleinrock  
*AFIPS '68 (Spring), April 1968*  
*An excellent early introduction to and analysis of a number of basic scheduling disciplines.*
- [J91] "The Art of Computer Systems Performance Analysis:  
 Techniques for Experimental Design, Measurement, Simulation, and Modeling"  
 R. Jain  
*Interscience, New York, April 1991*  
*The standard text on computer systems measurement. A great reference for your library, for sure.*
- [O45] "Animal Farm"  
 George Orwell  
*Secker and Warburg (London), 1945*  
*A great but depressing allegorical book about power and its corruptions. Some say it is a critique of Stalin and the pre-WWII Stalin era in the U.S.S.R; we say it's a critique of pigs.*
- [PV56] "Machine Repair as a Priority Waiting-Line Problem"  
 Thomas E. Phipps Jr. and W. R. Van Voorhis  
*Operations Research, 4:1, pages 76–86, February 1956*  
*Follow-on work that generalizes the SJF approach to machine repair from Cobham's original work; also postulates the utility of an STCF approach in such an environment. Specifically, "There are certain types of repair work, ... involving much dismantling and covering the floor with nuts and bolts, which certainly should not be interrupted once undertaken; in other cases it would be inadvisable to continue work on a long job if one or more short ones became available (p.81)."*
- [MB91] "The effect of context switches on cache performance"  
 Jeffrey C. Mogul and Anita Borg  
*ASPLOS, 1991*  
*A nice study on how cache performance can be affected by context switching; less of an issue in today's systems where processors issue billions of instructions per second but context-switches still happen in the millisecond time range.*
- [W15] "You can't have your cake and eat it"  
[http://en.wikipedia.org/wiki/You\\_can't\\_have\\_your\\_cake\\_and\\_eat\\_it](http://en.wikipedia.org/wiki/You_can't_have_your_cake_and_eat_it)  
*Wikipedia, as of December 2015*  
*The best part of this page is reading all the similar idioms from other languages. In Tamil, you can't "have both the moustache and drink the soup."*

## Homework

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

### Questions

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

## 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.

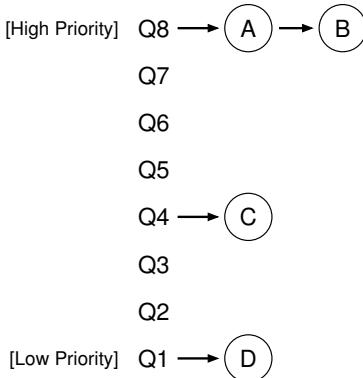
Thus, the key to MLFQ scheduling lies in how the scheduler sets priorities. Rather than giving a fixed priority to each job, MLFQ *varies* the priority of a job based on its *observed behavior*. If, for example, a job repeatedly relinquishes the CPU while waiting for input from the keyboard, MLFQ will 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.

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

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

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

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

Figure 8.1: **MLFQ Example**

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

## 8.2 Attempt #1: How To Change Priority

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

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

### Example 1: A Single Long-Running Job

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

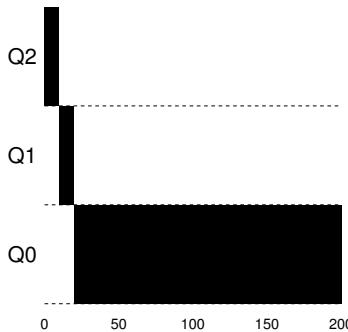


Figure 8.2: **Long-running Job Over Time**

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

### Example 2: Along Came A Short Job

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

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

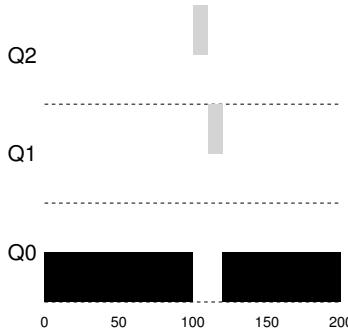


Figure 8.3: **Along Came An Interactive Job**

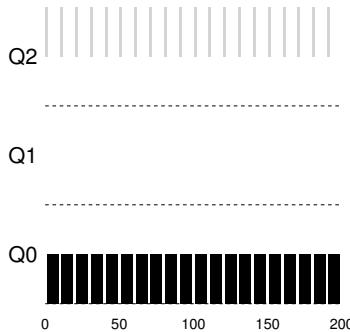


Figure 8.4: A Mixed I/O-intensive and CPU-intensive Workload

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

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

### Example 3: What About I/O?

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

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

### Problems With Our Current MLFQ

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

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

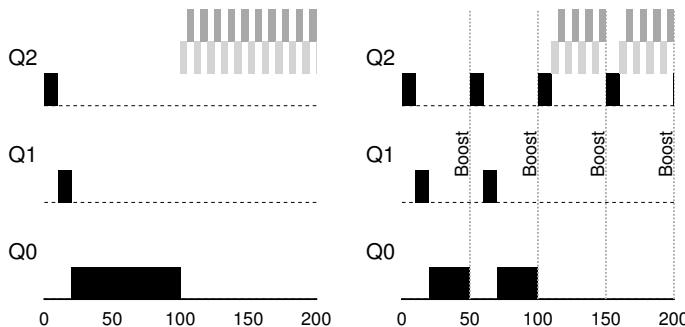


Figure 8.5: Without (Left) and With (Right) Priority Boost

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

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

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

### 8.3 Attempt #2: The Priority Boost

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

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

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

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

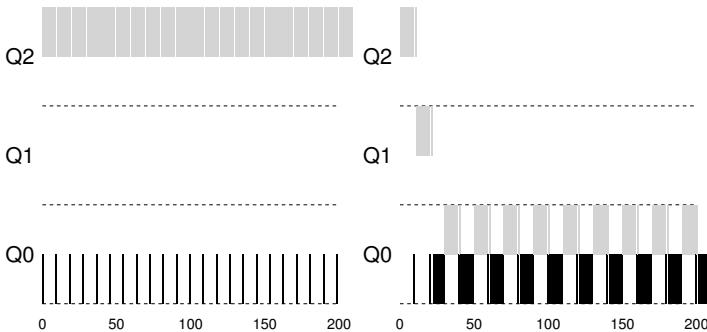


Figure 8.6: Without (Left) and With (Right) Gaming Tolerance

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

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

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

## 8.4 Attempt #3: Better Accounting

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

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

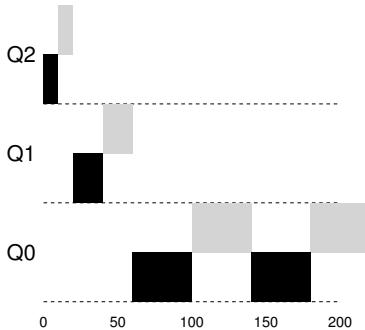


Figure 8.7: **Lower Priority, Longer Quanta**

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

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

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

## 8.5 Tuning MLFQ And Other Issues

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

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

TIP: AVOID VOO-DOO CONSTANTS (OUSTERHOUT'S LAW)

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

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

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

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

## 8.6 MLFQ: Summary

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

## TIP: USE ADVICE WHERE POSSIBLE

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

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

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

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

## References

[AD00] "Multilevel Feedback Queue Scheduling in Solaris"

Andrea Arpacı-Dusseau

Available: <http://www.cs.wisc.edu/~remzi/solaris-notes.pdf>

*A great short set of notes by one of the authors on the details of the Solaris scheduler. OK, we are probably biased in this description, but the notes are pretty darn good.*

[B86] "The Design of the UNIX Operating System"

M.J. Bach

Prentice-Hall, 1986

*One of the classic old books on how a real UNIX operating system is built; a definite must-read for kernel hackers.*

[C+62] "An Experimental Time-Sharing System"

F. J. Corbató, M. M. Daggett, R. C. Daley

IFIPS 1962

*A bit hard to read, but the source of many of the first ideas in multi-level feedback scheduling. Much of this later went into Multics, which one could argue was the most influential operating system of all time.*

[CS97] "Inside Windows NT"

Helen Custer and David A. Solomon

Microsoft Press, 1997

*The NT book, if you want to learn about something other than UNIX. Of course, why would you? OK, we're kidding; you might actually work for Microsoft some day you know.*

[E95] "An Analysis of Decay-Usage Scheduling in Multiprocessors"

D.H.J. Epema

SIGMETRICS '95

*A nice paper on the state of the art of scheduling back in the mid 1990s, including a good overview of the basic approach behind decay-usage schedulers.*

[LM+89] "The Design and Implementation of the 4.3BSD UNIX Operating System"

S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman

Addison-Wesley, 1989

*Another OS classic, written by four of the main people behind BSD. The later versions of this book, while more up to date, don't quite match the beauty of this one.*

[M06] "Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture"

Richard McDougall

Prentice-Hall, 2006

*A good book about Solaris and how it works.*

[O11] "John Ousterhout's Home Page"

John Ousterhout

Available: <http://www.stanford.edu/~ouster/>

*The home page of the famous Professor Ousterhout. The two co-authors of this book had the pleasure of taking graduate operating systems from Ousterhout while in graduate school; indeed, this is where the two co-authors got to know each other, eventually leading to marriage, kids, and even this book. Thus, you really can blame Ousterhout for this entire mess you're in.*

[P+95] "Informed Prefetching and Caching"

R.H. Patterson, G.A. Gibson, E. Grinting, D. Stodolsky, J. Zelenka

SOSP '95

*A fun paper about some very cool ideas in file systems, including how applications can give the OS advice about what files it is accessing and how it plans to access them.*

## Homework

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

## Questions

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

# Real Time Scheduling

## Introduction

Priority based scheduling enables us to give better service to certain processes. In our discussion of multi-queue scheduling, priority was adjusted based on whether a task was more interactive or compute intensive. But most schedulers enable us to give any process any desired priority. Isn't that good enough?

Priority scheduling is inherently a *best effort* approach. If our task is competing with other high priority tasks, it may not get as much time as it requires. Sometimes best effort isn't good enough:

- During reentry, the space shuttle is aerodynamically unstable. It is not actually being kept under control by the quick reflexes of the well-trained pilots, but rather by guidance computers that are collecting attitude and acceleration input and adjusting numerous spoilers hundreds of times per second.
- Scientific and military satellites may receive precious and irreplaceable sensor data at extremely high speeds. If it takes us too long to receive, process, and store one data frame, the next data frame may be lost.
- More mundanely, but also important, many manufacturing processes are run by computers nowadays. An assembly line needs to move at a particular speed, with each step being performed at a particular time. Performing the action too late results in a flawed or useless product.
- Even more commonly, playing media, like video or audio, has real time requirements. Sound must be produced at a certain rate and frames must be displayed frequently enough or the media becomes uncomfortable to deal with.

There are many computer controlled applications where delays in critical processing can have undesirable, or even disastrous consequences.

## What are Real-Time Systems

A real-time system is one whose correctness depends on timing as well as functionality.

When we discussed more traditional scheduling algorithms, the metrics we looked at were *turn-around time* (or throughput), *fairness*, and mean *response time*. But real-time systems have very different requirements, characterized by different metrics:

- *timeliness* ... how closely does it meet its timing requirements (e.g. ms/day of accumulated tardiness)
- *predictability* ... how much deviation is there in delivered timeliness

And we introduce a few new concepts:

- *feasibility* ... whether or not it is possible to meet the requirements for a particular task set
- *hard real-time* ... there are strong requirements that specified tasks be run a specified intervals (or within a specified response time). Failure to meet this requirement (perhaps by as little as a fraction of a micro-second) may result in system failure.
- *soft real-time* ... we may want to provide very good (e.g. microseconds) response time, the only consequences of missing a deadline are degraded performance or recoverable failures.

It sounds like real-time scheduling is more critical and difficult than traditional time-sharing, and in many ways it is. But real-time systems may have a few characteristics that make scheduling easier:

- We may actually know how long each task will take to run. This enables much more intelligent scheduling.

- *Starvation* (of low priority tasks) may be acceptable. The space shuttle absolutely must sense attitude and acceleration and adjust spoiler positions once per millisecond. But it probably doesn't matter if we update the navigational display once per millisecond or once every ten seconds. Telemetry transmission is probably somewhere in-between. Understanding the relative criticality of each task gives us the freedom to intelligently shed less critical work in times of high demand.
- The work-load may be relatively fixed. Normally high utilization implies long queuing delays, as bursty traffic creates long lines. But if the incoming traffic rate is relatively constant, it is possible to simultaneously achieve high utilization and good response time.

## Real-Time Scheduling Algorithms

In the simplest real-time systems, where the tasks and their execution times are all known, there might not even be a scheduler. One task might simply call (or yield to) the next. This model makes a great deal of sense in a system where the tasks form a producer/consumer pipeline (e.g. MPEG frame receipt, protocol decoding, image decompression, display).

In more complex real-time system, with a larger (but still fixed) number of tasks that do not function in a strictly pipeline fashion, it may be possible to do *static* scheduling. Based on the list of tasks to be run, and the expected completion time for each, we can define (at design or build time) a fixed schedule that will ensure timely execution of all tasks.

But for many real-time systems, the work-load changes from moment to moment, based on external events. These require *dynamic* scheduling. For *dynamic* scheduling algorithms, there are two key questions:

1. how they choose the next (ready) task to run
  - shortest job first
  - static priority ... highest priority ready task
  - soonest start-time deadline first (ASAP)
  - soonest completion-time deadline first (slack time)
2. how they handle overload (infeasible requirements)
  - best effort
  - periodicity adjustments ... run lower priority tasks less often.
  - work shedding ... stop running lower priority tasks entirely.

Preemption may also be a different issue in real-time systems. In ordinary time-sharing, preemption is a means of improving mean response time by breaking up the execution of long-running, compute-intensive tasks. A second advantage of preemptive scheduling, particularly important in a general purpose timesharing system, is that it prevents a buggy (infinite loop) program from taking over the CPU. The trade-off, between improved response time and increased overhead (for the added context switches), almost always favors preemptive scheduling. This may not be true for real-time systems:

- preempting a running task will almost surely cause it to miss its completion deadline.
- since we so often know what the expected execution time for a task will be, we can schedule accordingly and should have little need for preemption.
- embedded and real-time systems run fewer and simpler tasks than general purpose time systems, and the code is often much better tested ... so infinite loop bugs are extremely rare.

For the least demanding real time tasks, a sufficiently lightly loaded system might be reasonably successful in meeting its deadlines. However, this is achieved simply because the frequency at which the task is run happens to be high enough to meet its real time requirements, not because the scheduler is aware of such requirements. A lightly loaded machine running a traditional scheduler can often display a video to a user's satisfaction, not because the scheduler "knows" that a frame must be rendered by a certain deadline, but simply because the machine has enough cycles and a low enough work load to render the frame before the deadline has arrived

## Real-Time and Linux

Linux was not designed to be an embedded or real-time operating system, but many tasks that were once-considered embedded applications now require the capabilities (e.g. file systems, network protocols) of a general purpose operating system. As these requirements have increased and processors have gotten faster, increasingly many embedded and real-time applications have moved to Linux.

To support these applications Linux now supports a real-time scheduler, which can be enabled with [sched\\_setscheduler\(2\)](#), and is described in this [Linux Journal Article](#). This real-time scheduler does not provide quite the same level of response-time guarantees that more traditional Real-Time-OSs do, but they are adequate for many soft real-time applications.

What about Windows? Conventional wisdom states that Windows is not well suited for real time needs, offering no native real time scheduler and too many ways in which desired real time deadlines might be missed. Windows favors general purpose throughput over meeting deadlines, as a rule. With sufficiently low load, a Windows system may, nonetheless, provide fast enough service for some soft real time requirements, such as playing music or video. One should be careful in relying on Windows for critical real time operations, however, as it is not designed for that purpose.



## 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:** Hmm.... 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!

## The Abstraction: Address Spaces

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

### 13.1 Early Systems

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

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

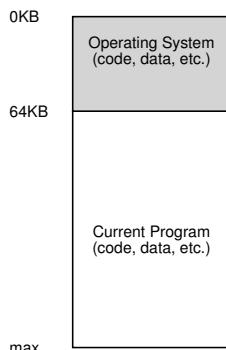


Figure 13.1: **Operating Systems: The Early Days**

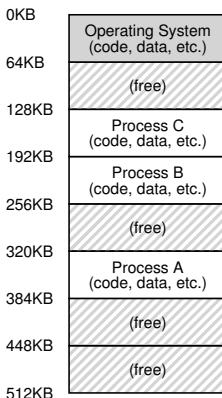


Figure 13.2: Three Processes: Sharing Memory

## 13.2 Multiprogramming and Time Sharing

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

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

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

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

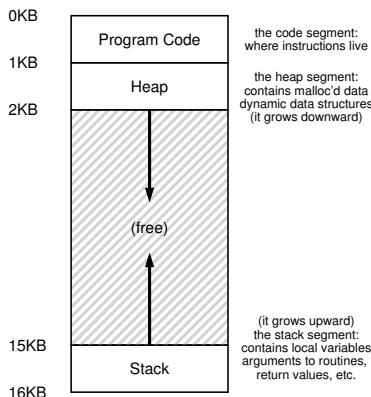


Figure 13.3: An Example Address Space

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

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

### 13.3 The Address Space

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

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

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

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

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

#### THE CRUX: HOW TO VIRTUALIZE MEMORY

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

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

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

---

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

**TIP: THE PRINCIPLE OF ISOLATION**

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

## 13.4 Goals

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

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

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

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

---

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

### ASIDE: EVERY ADDRESS YOU SEE IS VIRTUAL

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

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

```

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

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

```

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

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

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

---

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

## 13.5 Summary

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

## References

- [BH70] "The Nucleus of a Multiprogramming System"  
 Per Brinch Hansen  
*Communications of the ACM, 13:4, April 1970*  
*The first paper to suggest that the OS, or kernel, should be a minimal and flexible substrate for building customized operating systems; this theme is revisited throughout OS research history.*
- [CV65] "Introduction and Overview of the Multics System"  
 F. J. Corbato and V. A. Vyssotsky  
*Fall Joint Computer Conference, 1965*  
*A great early Multics paper. Here is the great quote about time sharing: "The impetus for time-sharing first arose from professional programmers because of their constant frustration in debugging programs at batch processing installations. Thus, the original goal was to time-share computers to allow simultaneous access by several persons while giving to each of them the illusion of having the whole machine at his disposal."*
- [DV66] "Programming Semantics for Multiprogrammed Computations"  
 Jack B. Dennis and Earl C. Van Horn  
*Communications of the ACM, Volume 9, Number 3, March 1966*  
*An early paper (but not the first) on multiprogramming.*
- [L60] "Man-Computer Symbiosis"  
 J. C. R. Licklider  
*IRE Transactions on Human Factors in Electronics, HFE-1:1, March 1960*  
*A funky paper about how computers and people are going to enter into a symbiotic age; clearly well ahead of its time but a fascinating read nonetheless.*
- [M62] "Time-Sharing Computer Systems"  
 J. McCarthy  
*Management and the Computer of the Future, MIT Press, Cambridge, Mass, 1962*  
*Probably McCarthy's earliest recorded paper on time sharing. However, in another paper [M83], he claims to have been thinking of the idea since 1957. McCarthy left the systems area and went on to become a giant in Artificial Intelligence at Stanford, including the creation of the LISP programming language. See McCarthy's home page for more info: <http://www-formal.stanford.edu/jmc/>*
- [M+63] "A Time-Sharing Debugging System for a Small Computer"  
 J. McCarthy, S. Boilen, E. Fredkin, J. C. R. Licklider  
*AFIPS '63 (Spring), May, 1963, New York, USA*  
*A great early example of a system that swapped program memory to the "drum" when the program wasn't running, and then back into "core" memory when it was about to be run.*
- [M83] "Reminiscences on the History of Time Sharing"  
 John McCarthy  
*Winter or Spring of 1983*  
*Available: <http://www-formal.stanford.edu/jmc/history/timesharing/timesharing.html>*  
*A terrific historical note on where the idea of time-sharing might have come from, including some doubts towards those who cite Strachey's work [S59] as the pioneering work in this area.*
- [R+89] "Mach: A System Software kernel"  
 Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alessandro Forin, David Golub, Michael Jones  
*COMPCON 89, February 1989*  
*Although not the first project on microkernels per se, the Mach project at CMU was well-known and influential; it still lives today deep in the bowels of Mac OS X.*

[S59] "Time Sharing in Large Fast Computers"

C. Strachey

Proceedings of the International Conference on Information Processing, UNESCO, June 1959

*One of the earliest references on time sharing.*

[S+03] "Improving the Reliability of Commodity Operating Systems"

Michael M. Swift, Brian N. Bershad, Henry M. Levy

SOSP 2003

*The first paper to show how microkernel-like thinking can improve operating system reliability.*

## 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<sup>1</sup>. The main problem we address is this:

### CRUX: HOW TO ALLOCATE AND MANAGE MEMORY

In UNIX/C programs, understanding how to allocate and manage memory is critical in building robust and reliable software. What interfaces are commonly used? What mistakes should be avoided?

### 14.1 Types of Memory

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

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

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

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

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

---

<sup>1</sup>Indeed, we hope all chapters are! But this one is shorter and pointier, we think.

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

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

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

Because of its explicit nature, and because of its more varied usage, heap memory presents more challenges to both users and systems. Thus, it is the focus of the remainder of our discussion.

## 14.2 The `malloc()` Call

The `malloc()` call is quite simple: you pass it a size asking for some room on the heap, and it either succeeds and gives you back a pointer to the newly-allocated space, or fails and returns `NULL`<sup>2</sup>.

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, or the right type).

The single parameter `malloc()` takes is of type `size_t` which simply describes how many bytes you need. However, most programmers do not type in a number here directly (such as 10); indeed, it would be considered poor form to do so. Instead, various routines and macros are utilized. For example, to allocate space for a double-precision floating point value, you simply do this:

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

---

<sup>2</sup>Note that `NULL` in C isn't really anything special at all, just a macro for the value zero.

**TIP: WHEN IN DOUBT, TRY IT OUT**

If you aren't sure how some routine or operator you are using behaves, there is no substitute for simply trying it out and making sure it behaves as you expect. While reading the manual pages or other documentation is useful, how it works in practice is what matters. Write some code and test it! That is no doubt the best way to make sure your code behaves as you desire. Indeed, that is what we did to double-check the things we were saying about `sizeof()` were actually true!

Wow, that's lot of *double-ing*! This invocation of `malloc()` uses the `sizeof()` operator to request the right amount of space; in C, this is generally thought of as a *compile-time* operator, meaning that the actual size is known at *compile time* and thus a number (in this case, 8, for a `double`) is substituted as the argument to `malloc()`. For this reason, `sizeof()` is correctly thought of as an operator and not a function call (a function call would take place at run time).

You can also pass in the name of a variable (and not just a type) to `sizeof()`, but in some cases you may not get the desired results, so be careful. For example, let's look at the following code snippet:

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));
```

In the first line, we've declared space for an array of 10 integers, which is fine and dandy. However, when we use `sizeof()` in the next line, it returns a small value, such as 4 (on 32-bit machines) or 8 (on 64-bit machines). The reason is that in this case, `sizeof()` thinks we are simply asking how big a *pointer* to an integer is, not how much memory we have dynamically allocated. However, sometimes `sizeof()` does work as you might expect:

```
int x[10];
printf("%d\n", sizeof(x));
```

In this case, there is enough static information for the compiler to know that 40 bytes have been allocated.

Another place to be careful is with strings. When declaring space for a string, use the following idiom: `malloc(strlen(s) + 1)`, which gets the length of the string using the function `strlen()`, and adds 1 to it in order to make room for the end-of-string character. Using `sizeof()` may lead to trouble here.

You might also notice that `malloc()` returns a pointer to type `void`. Doing so is just the way in C to pass back an address and let the programmer decide what to do with it. The programmer further helps out by using what is called a *cast*; in our example above, the programmer casts the return type of `malloc()` to a pointer to a `double`. Casting doesn't really accomplish anything, other than tell the compiler and other

programmers who might be reading your code: “yeah, I know what I’m doing.” By casting the result of `malloc()`, the programmer is just giving some reassurance; the cast is not needed for the correctness.

### 14.3 The `free()` Call

As it turns out, allocating memory is the easy part of the equation; knowing when, how, and even if to free memory is the hard part. To free heap memory that is no longer in use, programmers simply call `free()`:

```
int *x = malloc(10 * sizeof(int));
...
free(x);
```

The routine takes one argument, a pointer that was returned by `malloc()`. Thus, you might notice, the size of the allocated region is not passed in by the user, and must be tracked by the memory-allocation library itself.

### 14.4 Common Errors

There are a number of common errors that arise in the use of `malloc()` and `free()`. Here are some we’ve seen over and over again in teaching the undergraduate operating systems course. All of these examples compile and run with nary a peep from the compiler; while compiling a C program is necessary to build a correct C program, it is far from sufficient, as you will learn (often in the hard way).

Correct memory management has been such a problem, in fact, that many newer languages have support for **automatic memory management**. In such languages, while you call something akin to `malloc()` to allocate memory (usually `new` or something similar to allocate a new object), you never have to call something to free space; rather, a **garbage collector** runs and figures out what memory you no longer have references to and frees it for you.

#### Forgetting To Allocate Memory

Many routines expect memory to be allocated before you call them. For example, the routine `strcpy(dst, src)` copies a string from a source pointer to a destination pointer. However, if you are not careful, you might do this:

```
char *src = "hello";
char *dst;           // oops! unallocated
strcpy(dst, src);  // segfault and die
```

When you run this code, it will likely lead to a **segmentation fault**<sup>3</sup>, which is a fancy term for **YOU DID SOMETHING WRONG WITH MEMORY YOU FOOLISH PROGRAMMER AND I AM ANGRY.**

---

<sup>3</sup>Although it sounds arcane, you will soon learn why such an illegal memory access is called a segmentation fault; if that isn’t incentive to read on, what is?

**TIP: IT COMPILED OR IT RAN  $\neq$  IT IS CORRECT**

Just because a program compiled(!) or even ran once or many times correctly does not mean the program is correct. Many events may have conspired to get you to a point where you believe it works, but then something changes and it stops. A common student reaction is to say (or yell) "But it worked before!" and then blame the compiler, operating system, hardware, or even (dare we say it) the professor. But the problem is usually right where you think it would be, in your code. Get to work and debug it before you blame those other components.

In this case, the proper code might instead look like this:

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src) + 1);
strcpy(dst, src); // work properly
```

Alternately, you could use `strdup()` and make your life even easier. Read the `strdup` man page for more information.

## Not Allocating Enough Memory

A related error is not allocating enough memory, sometimes called a **buffer overflow**. In the example above, a common error is to make *almost* enough room for the destination buffer.

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src)); // too small!
strcpy(dst, src); // work properly
```

Oddly enough, depending on how `malloc` is implemented and many other details, this program will often run seemingly correctly. In some cases, when the string copy executes, it writes one byte too far past the end of the allocated space, but in some cases this is harmless, perhaps overwriting a variable that isn't used anymore. In some cases, these overflows can be incredibly harmful, and in fact are the source of many security vulnerabilities in systems [W06]. In other cases, the `malloc` library allocated a little extra space anyhow, and thus your program actually doesn't scribble on some other variable's value and works quite fine. In even other cases, the program will indeed fault and crash. And thus we learn another valuable lesson: even though it ran correctly once, doesn't mean it's correct.

## Forgetting to Initialize Allocated Memory

With this error, you call `malloc()` properly, but forget to fill in some values into your newly-allocated data type. Don't do this! If you do forget, your program will eventually encounter an **uninitialized read**, where it

reads from the heap some data of unknown value. Who knows what might be in there? If you're lucky, some value such that the program still works (e.g., zero). If you're not lucky, something random and harmful.

### Forgetting To Free Memory

Another common error is known as a **memory leak**, and it occurs when you forget to free memory. In long-running applications or systems (such as the OS itself), this is a huge problem, as slowly leaking memory eventually leads one to run out of memory, at which point a restart is required. Thus, in general, when you are done with a chunk of memory, you should make sure to free it. Note that using a garbage-collected language doesn't help here: if you still have a reference to some chunk of memory, no garbage collector will ever free it, and thus memory leaks remain a problem even in more modern languages.

In some cases, it may seem like not calling `free()` is reasonable. For example, your program is short-lived, and will soon exit; in this case, when the process dies, the OS will clean up all of its allocated pages and thus no memory leak will take place per se. While this certainly "works" (see the aside on page 7), it is probably a bad habit to develop, so be wary of choosing such a strategy. In the long run, one of your goals as a programmer is to develop good habits; one of those habits is understanding how you are managing memory, and (in languages like C), freeing the blocks you have allocated. Even if you can get away with not doing so, it is probably good to get in the habit of freeing each and every byte you explicitly allocate.

### Freeing Memory Before You Are Done With It

Sometimes a program will free memory before it is finished using it; such a mistake is called a **dangling pointer**, and it, as you can guess, is also a bad thing. The subsequent use can crash the program, or overwrite valid memory (e.g., you called `free()`, but then called `malloc()` again to allocate something else, which then recycles the errantly-freed memory).

### Freeing Memory Repeatedly

Programs also sometimes free memory more than once; this is known as the **double free**. The result of doing so is undefined. As you can imagine, the memory-allocation library might get confused and do all sorts of weird things; crashes are a common outcome.

### Calling `free()` Incorrectly

One last problem we discuss is the call of `free()` incorrectly. After all, `free()` expects you only to pass to it one of the pointers you received from `malloc()` earlier. When you pass in some other value, bad things can (and do) happen. Thus, such **invalid frees** are dangerous and of course should also be avoided.

**ASIDE: WHY NO MEMORY IS LEAKED ONCE YOUR PROCESS EXITS**

When you write a short-lived program, you might allocate some space using `malloc()`. The program runs and is about to complete: is there need to call `free()` a bunch of times just before exiting? While it seems wrong not to, no memory will be “lost” in any real sense. The reason is simple: there are really two levels of memory management in the system.

The first level of memory management is performed by the OS, which hands out memory to processes when they run, and takes it back when processes exit (or otherwise die). The second level of management is *within* each process, for example within the heap when you call `malloc()` and `free()`. Even if you fail to call `free()` (and thus leak memory in the heap), the operating system will reclaim *all* the memory of the process (including those pages for code, stack, and, as relevant here, heap) when the program is finished running. No matter what the state of your heap in your address space, the OS takes back all of those pages when the process dies, thus ensuring that no memory is lost despite the fact that you didn’t free it.

Thus, for short-lived programs, leaking memory often does not cause any operational problems (though it may be considered poor form). When you write a long-running server (such as a web server or database management system, which never exit), leaked memory is a much bigger issue, and will eventually lead to a crash when the application runs out of memory. And of course, leaking memory is an even larger issue inside one particular program: the operating system itself. Showing us once again: those who write the kernel code have the toughest job of all...

## Summary

As you can see, there are lots of ways to abuse memory. Because of frequent errors with memory, a whole ecosystem of tools have developed to help find such problems in your code. Check out both **purify** [HJ92] and **valgrind** [SN05]; both are excellent at helping you locate the source of your memory-related problems. Once you become accustomed to using these powerful tools, you will wonder how you survived without them.

## 14.5 Underlying OS Support

You might have noticed that we haven’t been talking about system calls when discussing `malloc()` and `free()`. The reason for this is simple: they are not system calls, but rather library calls. Thus the malloc library manages space within your virtual address space, but itself is built on top of some system calls which call into the OS to ask for more memory or release some back to the system.

One such system call is called `brk`, which is used to change the location of the program’s **break**: the location of the end of the heap. It takes one argument (the address of the new break), and thus either increases or decreases the size of the heap based on whether the new break is larger or smaller than the current break. An additional call `sbrk` is passed an increment but otherwise serves a similar purpose.

Note that you should never directly call either `brk` or `sbrk`. They are used by the memory-allocation library; if you try to use them, you will likely make something go (horribly) wrong. Stick to `malloc()` and `free()` instead.

Finally, you can also obtain memory from the operating system via the `mmap()` call. By passing in the correct arguments, `mmap()` can create an **anonymous** memory region within your program — a region which is not associated with any particular file but rather with **swap space**, something we’ll discuss in detail later on in virtual memory. This memory can then also be treated like a heap and managed as such. Read the manual page of `mmap()` for more details.

## 14.6 Other Calls

There are a few other calls that the memory-allocation library supports. For example, `calloc()` allocates memory and also zeroes it before returning; this prevents some errors where you assume that memory is zeroed and forget to initialize it yourself (see the paragraph on “uninitialized reads” above). The routine `realloc()` can also be useful, when you’ve allocated space for something (say, an array), and then need to add something to it: `realloc()` makes a new larger region of memory, copies the old region into it, and returns the pointer to the new region.

## 14.7 Summary

We have introduced some of the APIs dealing with memory allocation. As always, we have just covered the basics; more details are available elsewhere. Read the C book [KR88] and Stevens [SR05] (Chapter 7) for more information. For a cool modern paper on how to detect and correct many of these problems automatically, see Novark et al. [N+07]; this paper also contains a nice summary of common problems and some neat ideas on how to find and fix them.

## References

- [HJ92] Purify: Fast Detection of Memory Leaks and Access Errors  
R. Hastings and B. Joyce  
USENIX Winter '92  
*The paper behind the cool Purify tool, now a commercial product.*
- [KR88] "The C Programming Language"  
Brian Kernighan and Dennis Ritchie  
Prentice-Hall 1988  
*The C book, by the developers of C. Read it once, do some programming, then read it again, and then keep it near your desk or wherever you program.*
- [N+07] "Exterminator: Automatically Correcting Memory Errors with High Probability"  
Gene Novark, Emery D. Berger, and Benjamin G. Zorn  
PLDI 2007  
*A cool paper on finding and correcting memory errors automatically, and a great overview of many common errors in C and C++ programs.*
- [SN05] "Using Valgrind to Detect Undefined Value Errors with Bit-precision"  
J. Seward and N. Nethercote  
USENIX '05  
*How to use valgrind to find certain types of errors.*
- [SR05] "Advanced Programming in the UNIX Environment"  
W. Richard Stevens and Stephen A. Rago  
Addison-Wesley, 2005  
*We've said it before, we'll say it again: read this book many times and use it as a reference whenever you are in doubt. The authors are always surprised at how each time they read something in this book, they learn something new, even after many years of C programming.*
- [W06] "Survey on Buffer Overflow Attacks and Countermeasures"  
Tim Werthman  
Available: [www.nds.rub.de/lehre/seminar/SS06/Werthmann\\_BufferOverflow.pdf](http://www.nds.rub.de/lehre/seminar/SS06/Werthmann_BufferOverflow.pdf)  
*A nice survey of buffer overflows and some of the security problems they cause. Refers to many of the famous exploits.*

## Homework (Code)

In this homework, you will gain some familiarity with memory allocation. First, you'll write some buggy programs (fun!). Then, you'll use some tools to help you find the bugs you inserted. Then, you will realize how awesome these tools are and use them in the future, thus making yourself more happy and productive.

The first tool you'll use is `gdb`, the debugger. There is a lot to learn about this debugger; here we'll only scratch the surface.

The second tool you'll use is `valgrind` [SN05]. This tool helps find memory leaks and other insidious memory problems in your program. If it's not installed on your system, go to the website and do so:

<http://valgrind.org/downloads/current.html>

## Questions

1. First, write a simple program called `null.c` that creates a pointer to an integer, sets it to `NULL`, and then tries to dereference it. Compile this into an executable called `null`. What happens when you run this program?
2. Next, compile this program with symbol information included (with the `-g` flag). Doing so let's put more information into the executable, enabling the debugger to access more useful information about variable names and the like. Run the program under the debugger by typing `gdb null` and then, once `gdb` is running, typing `run`. What does `gdb` show you?
3. Finally, use the `valgrind` tool on this program. We'll use the `memcheck` tool that is a part of `valgrind` to analyze what happens. Run this by typing in the following: `valgrind --leak-check=yes null`. What happens when you run this? Can you interpret the output from the tool?
4. Write a simple program that allocates memory using `malloc()` but forgets to free it before exiting. What happens when this program runs? Can you use `gdb` to find any problems with it? How about `valgrind` (again with the `--leak-check=yes` flag)?
5. Write a program that creates an array of integers called `data` of size 100 using `malloc`; then, set `data[100]` to zero. What happens when you run this program? What happens when you run this program using `valgrind`? Is the program correct?
6. Create a program that allocates an array of integers (as above), frees them, and then tries to print the value of one of the elements of the array. Does the program run? What happens when you use `valgrind` on it?
7. Now pass a funny value to free (e.g., a pointer in the middle of the array you allocated above). What happens? Do you need tools to find this type of problem?

8. Try out some of the other interfaces to memory allocation. For example, create a simple vector-like data structure and related routines that use `realloc()` to manage the vector. Use an array to store the vectors elements; when a user adds an entry to the vector, use `realloc()` to allocate more space for it. How well does such a vector perform? How does it compare to a linked list? Use `valgrind` to help you find bugs.
9. Spend more time and read about using `gdb` and `valgrind`. Knowing your tools is critical; spend the time and learn how to become an expert debugger in the UNIX and C environment.

## Mechanism: Address Translation

In developing the virtualization of the CPU, we focused on a general mechanism known as **limited direct execution** (or LDE). The idea behind LDE is simple: for the most part, let the program run directly on the hardware; however, at certain key points in time (such as when a process issues a system call, or a timer interrupt occurs), arrange so that the OS gets involved and makes sure the “right” thing happens. Thus, the OS, with a little hardware support, tries its best to get out of the way of the running program, to deliver an *efficient* virtualization; however, by **interposing** at those critical points in time, the OS ensures that it maintains *control* over the hardware. Efficiency and control together are two of the main goals of any modern operating system.

In virtualizing memory, we will pursue a similar strategy, attaining both efficiency and control while providing the desired virtualization. Efficiency dictates that we make use of hardware support, which at first will be quite rudimentary (e.g., just a few registers) but will grow to be fairly complex (e.g., TLBs, page-table support, and so forth, as you will see). Control implies that the OS ensures that no application is allowed to access any memory but its own; thus, to protect applications from one another, and the OS from applications, we will need help from the hardware here too. Finally, we will need a little more from the VM system, in terms of *flexibility*; specifically, we’d like for programs to be able to use their address spaces in whatever way they would like, thus making the system easier to program. And thus we arrive at the refined crux:

### THE CRUX: HOW TO EFFICIENTLY AND FLEXIBLY VIRTUALIZE MEMORY

How can we build an efficient virtualization of memory? How do we provide the flexibility needed by applications? How do we maintain control over which memory locations an application can access, and thus ensure that application memory accesses are properly restricted? How do we do all of this efficiently?

The generic technique we will use, which you can consider an addition to our general approach of limited direct execution, is something that is referred to as **hardware-based address translation**, or just **address translation** for short. With address translation, the hardware transforms each memory access (e.g., an instruction fetch, load, or store), changing the **virtual** address provided by the instruction to a **physical** address where the desired information is actually located. Thus, on each and every memory reference, an address translation is performed by the hardware to redirect application memory references to their actual locations in memory.

Of course, the hardware alone cannot virtualize memory, as it just provides the low-level mechanism for doing so efficiently. The OS must get involved at key points to set up the hardware so that the correct translations take place; it must thus **manage memory**, keeping track of which locations are free and which are in use, and judiciously intervening to maintain control over how memory is used.

Once again the goal of all of this work is to create a beautiful **illusion**: that the program has its own private memory, where its own code and data reside. Behind that virtual reality lies the ugly physical truth: that many programs are actually sharing memory at the same time, as the CPU (or CPUs) switches between running one program and the next. Through virtualization, the OS (with the hardware's help) turns the ugly machine reality into something that is a useful, powerful, and easy to use abstraction.

## 15.1 Assumptions

Our first attempts at virtualizing memory will be very simple, almost laughably so. Go ahead, laugh all you want; pretty soon it will be the OS laughing at you, when you try to understand the ins and outs of TLBs, multi-level page tables, and other technical wonders. Don't like the idea of the OS laughing at you? Well, you may be out of luck then; that's just how the OS rolls.

Specifically, we will assume for now that the user's address space must be placed *contiguously* in physical memory. We will also assume, for simplicity, that the size of the address space is not too big; specifically, that it is *less than the size of physical memory*. Finally, we will also assume that each address space is exactly the *same size*. Don't worry if these assumptions sound unrealistic; we will relax them as we go, thus achieving a realistic virtualization of memory.

## 15.2 An Example

To understand better what we need to do to implement address translation, and why we need such a mechanism, let's look at a simple example. Imagine there is a process whose address space is as indicated in Figure 15.1. What we are going to examine here is a short code sequence

#### TIP: INTERPOSITION IS POWERFUL

Interposition is a generic and powerful technique that is often used to great effect in computer systems. In virtualizing memory, the hardware will interpose on each memory access, and translate each virtual address issued by the process to a physical address where the desired information is actually stored. However, the general technique of interposition is much more broadly applicable; indeed, almost any well-defined interface can be interposed upon, to add new functionality or improve some other aspect of the system. One of the usual benefits of such an approach is **transparency**; the interposition often is done without changing the client of the interface, thus requiring no changes to said client.

that loads a value from memory, increments it by three, and then stores the value back into memory. You can imagine the C-language representation of this code might look like this:

```
void func() {
    int x;
    x = x + 3; // this is the line of code we are interested in
```

The compiler turns this line of code into assembly, which might look something like this (in x86 assembly). Use `objdump` on Linux or `otool` on Mac OS X to disassemble it:

```
128: movl 0x0(%ebx), %eax      ;load 0+ebx into eax
132: addl $0x03, %eax         ;add 3 to eax register
135: movl %eax, 0x0(%ebx)     ;store eax back to mem
```

This code snippet is relatively straightforward; it presumes that the address of `x` has been placed in the register `ebx`, and then loads the value at that address into the general-purpose register `eax` using the `movl` instruction (for “longword” move). The next instruction adds 3 to `eax`, and the final instruction stores the value in `eax` back into memory at that same location.

In Figure 15.1 (page 4), you can see how both the code and data are laid out in the process’s address space; the three-instruction code sequence is located at address 128 (in the code section near the top), and the value of the variable `x` at address 15 KB (in the stack near the bottom). In the figure, the initial value of `x` is 3000, as shown in its location on the stack.

When these instructions run, from the perspective of the process, the following memory accesses take place.

- Fetch instruction at address 128
- Execute this instruction (load from address 15 KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

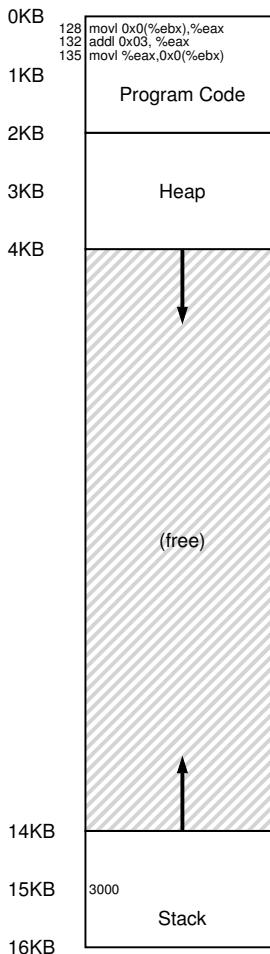


Figure 15.1: A Process And Its Address Space

From the program's perspective, its **address space** starts at address 0 and grows to a maximum of 16 KB; all memory references it generates should be within these bounds. However, to virtualize memory, the OS wants to place the process somewhere else in physical memory, not necessarily at address 0. Thus, we have the problem: how can we **relocate** this process in memory in a way that is **transparent** to the process? How can we provide the illusion of a virtual address space starting at 0, when in reality the address space is located at some other physical address?

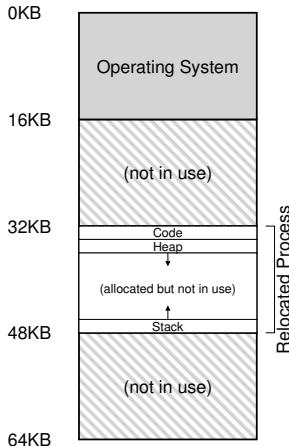


Figure 15.2: Physical Memory with a Single Relocated Process

An example of what physical memory might look like once this process's address space has been placed in memory is found in Figure 15.2. In the figure, you can see the OS using the first slot of physical memory for itself, and that it has relocated the process from the example above into the slot starting at physical memory address 32 KB. The other two slots are free (16 KB-32 KB and 48 KB-64 KB).

### 15.3 Dynamic (Hardware-based) Relocation

To gain some understanding of hardware-based address translation, we'll first discuss its first incarnation. Introduced in the first time-sharing machines of the late 1950's is a simple idea referred to as **base and bounds**; the technique is also referred to as **dynamic relocation**; we'll use both terms interchangeably [SS74].

Specifically, we'll need two hardware registers within each CPU: one is called the **base** register, and the other the **bounds** (sometimes called a **limit** register). This base-and-bounds pair is going to allow us to place the address space anywhere we'd like in physical memory, and do so while ensuring that the process can only access its own address space.

In this setup, each program is written and compiled as if it is loaded at address zero. However, when a program starts running, the OS decides where in physical memory it should be loaded and sets the base register to that value. In the example above, the OS decides to load the process at physical address 32 KB and thus sets the base register to this value.

Interesting things start to happen when the process is running. Now, when any memory reference is generated by the process, it is **translated** by the processor in the following manner:

$$\text{physical address} = \text{virtual address} + \text{base}$$

#### ASIDE: SOFTWARE-BASED RELOCATION

In the early days, before hardware support arose, some systems performed a crude form of relocation purely via software methods. The basic technique is referred to as **static relocation**, in which a piece of software known as the **loader** takes an executable that is about to be run and rewrites its addresses to the desired offset in physical memory.

For example, if an instruction was a load from address 1000 into a register (e.g., `movl 1000, %eax`), and the address space of the program was loaded starting at address 3000 (and not 0, as the program thinks), the loader would rewrite the instruction to offset each address by 3000 (e.g., `movl 4000, %eax`). In this way, a simple static relocation of the process's address space is achieved.

However, static relocation has numerous problems. First and most importantly, it does not provide protection, as processes can generate bad addresses and thus illegally access other process's or even OS memory; in general, hardware support is likely needed for true protection [WL+93]. Another negative is that once placed, it is difficult to later relocate an address space to another location [M65].

Each memory reference generated by the process is a **virtual address**; the hardware in turn adds the contents of the base register to this address and the result is a **physical address** that can be issued to the memory system.

To understand this better, let's trace through what happens when a single instruction is executed. Specifically, let's look at one instruction from our earlier sequence:

```
128: movl 0x0(%ebx), %eax
```

The program counter (PC) is set to 128; when the hardware needs to fetch this instruction, it first adds the value to the base register value of 32 KB (32768) to get a physical address of 32896; the hardware then fetches the instruction from that physical address. Next, the processor begins executing the instruction. At some point, the process then issues the load from virtual address 15 KB, which the processor takes and again adds to the base register (32 KB), getting the final physical address of 47 KB and thus the desired contents.

Transforming a virtual address into a physical address is exactly the technique we refer to as **address translation**; that is, the hardware takes a virtual address the process thinks it is referencing and transforms it into a physical address which is where the data actually resides. Because this relocation of the address happens at runtime, and because we can move address spaces even after the process has started running, the technique is often referred to as **dynamic relocation** [M65].

**TIP: HARDWARE-BASED DYNAMIC RELOCATION**

With dynamic relocation, a little hardware goes a long way. Namely, a **base** register is used to transform virtual addresses (generated by the program) into physical addresses. A **bounds** (or **limit**) register ensures that such addresses are within the confines of the address space. Together they provide a simple and efficient virtualization of memory.

Now you might be asking: what happened to that bounds (limit) register? After all, isn't this the base *and* bounds approach? Indeed, it is. As you might have guessed, the bounds register is there to help with protection. Specifically, the processor will first check that the memory reference is *within bounds* to make sure it is legal; in the simple example above, the bounds register would always be set to 16 KB. If a process generates a virtual address that is greater than the bounds, or one that is negative, the CPU will raise an exception, and the process will likely be terminated. The point of the bounds is thus to make sure that all addresses generated by the process are legal and within the "bounds" of the process.

We should note that the base and bounds registers are hardware structures kept on the chip (one pair per CPU). Sometimes people call the part of the processor that helps with address translation the **memory management unit (MMU)**; as we develop more sophisticated memory management techniques, we will be adding more circuitry to the MMU.

A small aside about bound registers, which can be defined in one of two ways. In one way (as above), it holds the *size* of the address space, and thus the hardware checks the virtual address against it first before adding the base. In the second way, it holds the *physical address* of the end of the address space, and thus the hardware first adds the base and then makes sure the address is within bounds. Both methods are logically equivalent; for simplicity, we'll usually assume the former method.

### Example Translations

To understand address translation via base-and-bounds in more detail, let's take a look at an example. Imagine a process with an address space of size 4 KB (yes, unrealistically small) has been loaded at physical address 16 KB. Here are the results of a number of address translations:

Virtual Address		Physical Address
0	→	16 KB
1 KB	→	17 KB
3000	→	19384
4400	→	<i>Fault (out of bounds)</i>

As you can see from the example, it is easy for you to simply add the base address to the virtual address (which can rightly be viewed as an *offset* into the address space) to get the resulting physical address. Only if the virtual address is "too big" or negative will the result be a fault, causing an exception to be raised.

**ASIDE: DATA STRUCTURE — THE FREE LIST**

The OS must track which parts of free memory are not in use, so as to be able to allocate memory to processes. Many different data structures can of course be used for such a task; the simplest (which we will assume here) is a **free list**, which simply is a list of the ranges of the physical memory which are not currently in use.

## 15.4 Hardware Support: A Summary

Let us now summarize the support we need from the hardware (also see Figure 15.3, page 9). First, as discussed in the chapter on CPU virtualization, we require two different CPU modes. The OS runs in **privileged mode** (or **kernel mode**), where it has access to the entire machine; applications run in **user mode**, where they are limited in what they can do. A single bit, perhaps stored in some kind of **processor status word**, indicates which mode the CPU is currently running in; upon certain special occasions (e.g., a system call or some other kind of exception or interrupt), the CPU switches modes.

The hardware must also provide the **base and bounds registers** themselves; each CPU thus has an additional pair of registers, part of the **memory management unit (MMU)** of the CPU. When a user program is running, the hardware will translate each address, by adding the base value to the virtual address generated by the user program. The hardware must also be able to check whether the address is valid, which is accomplished by using the bounds register and some circuitry within the CPU.

The hardware should provide special instructions to modify the base and bounds registers, allowing the OS to change them when different processes run. These instructions are **privileged**; only in kernel (or privileged) mode can the registers be modified. Imagine the havoc a user process could wreak<sup>1</sup> if it could arbitrarily change the base register while running. Imagine it! And then quickly flush such dark thoughts from your mind, as they are the ghastly stuff of which nightmares are made.

Finally, the CPU must be able to generate **exceptions** in situations where a user program tries to access memory illegally (with an address that is “out of bounds”); in this case, the CPU should stop executing the user program and arrange for the OS “out-of-bounds” **exception handler** to run. The OS handler can then figure out how to react, in this case likely terminating the process. Similarly, if a user program tries to change the values of the (privileged) base and bounds registers, the CPU should raise an exception and run the “tried to execute a privileged operation while in user mode” handler. The CPU also must provide a method to inform it of the location of these handlers; a few more privileged instructions are thus needed.

---

<sup>1</sup>Is there anything other than “havoc” that can be “wreaked”?

Hardware Requirements	Notes
Privileged mode	<i>Needed to prevent user-mode processes from executing privileged operations</i>
Base/bounds registers	<i>Need pair of registers per CPU to support address translation and bounds checks</i>
Ability to translate virtual addresses and check if within bounds	<i>Circuitry to do translations and check limits; in this case, quite simple</i>
Privileged instruction(s) to update base/bounds	<i>OS must be able to set these values before letting a user program run</i>
Privileged instruction(s) to register exception handlers	<i>OS must be able to tell hardware what code to run if exception occurs</i>
Ability to raise exceptions	<i>When processes try to access privileged instructions or out-of-bounds memory</i>

Figure 15.3: Dynamic Relocation: Hardware Requirements

## 15.5 Operating System Issues

Just as the hardware provides new features to support dynamic relocation, the OS now has new issues it must handle; the combination of hardware support and OS management leads to the implementation of a simple virtual memory. Specifically, there are a few critical junctures where the OS must get involved to implement our base-and-bounds version of virtual memory.

First, the OS must take action when a process is created, finding space for its address space in memory. Fortunately, given our assumptions that each address space is (a) smaller than the size of physical memory and (b) the same size, this is quite easy for the OS; it can simply view physical memory as an array of slots, and track whether each one is free or in use. When a new process is created, the OS will have to search a data structure (often called a **free list**) to find room for the new address space and then mark it used. With variable-sized address spaces, life is more complicated, but we will leave that concern for future chapters.

Let's look at an example. In Figure 15.2 (page 5), you can see the OS using the first slot of physical memory for itself, and that it has relocated the process from the example above into the slot starting at physical memory address 32 KB. The other two slots are free (16 KB-32 KB and 48 KB-64 KB); thus, the **free list** should consist of these two entries.

Second, the OS must do some work when a process is terminated (i.e., when it exits gracefully, or is forcefully killed because it misbehaved), reclaiming all of its memory for use in other processes or the OS. Upon termination of a process, the OS thus puts its memory back on the free list, and cleans up any associated data structures as need be.

Third, the OS must also perform a few additional steps when a context switch occurs. There is only one base and bounds register pair on each CPU, after all, and their values differ for each running program, as each program is loaded at a different physical address in memory. Thus, the OS must *save and restore* the base-and-bounds pair when it switches be-

OS Requirements	Notes
Memory management	<i>Need to allocate memory for new processes; Reclaim memory from terminated processes; Generally manage memory via free list</i>
Base/bounds management	<i>Must set base/bounds properly upon context switch</i>
Exception handling	<i>Code to run when exceptions arise; likely action is to terminate offending process</i>

Figure 15.4: Dynamic Relocation: Operating System Responsibilities

tween processes. Specifically, when the OS decides to stop running a process, it must save the values of the base and bounds registers to memory, in some per-process structure such as the **process structure** or **process control block** (PCB). Similarly, when the OS resumes a running process (or runs it the first time), it must set the values of the base and bounds on the CPU to the correct values for this process.

We should note that when a process is stopped (i.e., not running), it is possible for the OS to move an address space from one location in memory to another rather easily. To move a process's address space, the OS first deschedules the process; then, the OS copies the address space from the current location to the new location; finally, the OS updates the saved base register (in the process structure) to point to the new location. When the process is resumed, its (new) base register is restored, and it begins running again, oblivious that its instructions and data are now in a completely new spot in memory.

Fourth, the OS must provide **exception handlers**, or functions to be called, as discussed above; the OS installs these handlers at boot time (via privileged instructions). For example, if a process tries to access memory outside its bounds, the CPU will raise an exception; the OS must be prepared to take action when such an exception arises. The common reaction of the OS will be one of hostility: it will likely terminate the offending process. The OS should be highly protective of the machine it is running, and thus it does not take kindly to a process trying to access memory or execute instructions that it shouldn't. Bye bye, misbehaving process; it's been nice knowing you.

Figure 15.5 (page 11) illustrates much of the hardware/OS interaction in a timeline. The figure shows what the OS does at boot time to ready the machine for use, and then what happens when a process (Process A) starts running; note how its memory translations are handled by the hardware with no OS intervention. At some point, a timer interrupt occurs, and the OS switches to Process B, which executes a "bad load" (to an illegal memory address); at that point, the OS must get involved, terminating the process and cleaning up by freeing B's memory and removing its entry from the process table. As you can see from the diagram, we are still following the basic approach of **limited direct execution**. In most cases, the OS just sets up the hardware appropriately and lets the process run directly on the CPU; only when the process misbehaves does the OS have to become involved.

OS @ boot (kernel mode)	Hardware	
initialize trap table	remember addresses of... system call handler timer handler illegal mem-access handler illegal instruction handler	
start interrupt timer	start timer; interrupt after X ms	
initialize process table		
initialize free list		
OS @ run (kernel mode)	Hardware	Program (user mode)
To start process A: allocate entry in process table allocate memory for process set base/bounds registers <b>return-from-trap</b> (into A)	restore registers of A move to <b>user mode</b> jump to A's (initial) PC	Process A runs Fetch instruction
	Translate virtual address and perform fetch	Execute instruction
	If explicit load/store: Ensure address is in-bounds; Translate virtual address and perform load/store	
	...	
	<b>Timer interrupt</b> move to <b>kernel mode</b> Jump to interrupt handler	
<b>Handle the trap</b> Call <code>switch()</code> routine save regs(A) to proc-struct(A) (including base/bounds) restore regs(B) from proc-struct(B) (including base/bounds) <b>return-from-trap</b> (into B)	restore registers of B move to <b>user mode</b> jump to B's PC	Process B runs Execute bad load
	Load is out-of-bounds; move to <b>kernel mode</b> jump to trap handler	
<b>Handle the trap</b> Decide to terminate process B de-allocate B's memory free B's entry in process table		

Figure 15.5: Limited Direct Execution Protocol (Dynamic Relocation)

## 15.6 Summary

In this chapter, we have extended the concept of limited direct execution with a specific mechanism used in virtual memory, known as **address translation**. With address translation, the OS can control each and every memory access from a process, ensuring the accesses stay within the bounds of the address space. Key to the efficiency of this technique is hardware support, which performs the translation quickly for each access, turning virtual addresses (the process's view of memory) into physical ones (the actual view). All of this is performed in a way that is *transparent* to the process that has been relocated; the process has no idea its memory references are being translated, making for a wonderful illusion.

We have also seen one particular form of virtualization, known as base and bounds or dynamic relocation. Base-and-bounds virtualization is quite *efficient*, as only a little more hardware logic is required to add a base register to the virtual address and check that the address generated by the process is in bounds. Base-and-bounds also offers *protection*; the OS and hardware combine to ensure no process can generate memory references outside its own address space. Protection is certainly one of the most important goals of the OS; without it, the OS could not control the machine (if processes were free to overwrite memory, they could easily do nasty things like overwrite the trap table and take over the system).

Unfortunately, this simple technique of dynamic relocation does have its inefficiencies. For example, as you can see in Figure 15.2 (page 5), the relocated process is using physical memory from 32 KB to 48 KB; however, because the process stack and heap are not too big, all of the space between the two is simply *wasted*. This type of waste is usually called **internal fragmentation**, as the space *inside* the allocated unit is not all used (i.e., is fragmented) and thus wasted. In our current approach, although there might be enough physical memory for more processes, we are currently restricted to placing an address space in a fixed-sized slot and thus internal fragmentation can arise<sup>2</sup>. 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.

---

<sup>2</sup>A different solution might instead place a fixed-sized stack within the address space, just below the code region, and a growing heap below that. However, this limits flexibility by making recursion and deeply-nested function calls challenging, and thus is something we hope to avoid.

## References

- [M65] "On Dynamic Program Relocation"  
W.C. McGee  
IBM Systems Journal  
Volume 4, Number 3, 1965, pages 184–199  
*This paper is a nice summary of early work on dynamic relocation, as well as some basics on static relocation.*
- [P90] "Relocating loader for MS-DOS .EXE executable files"  
Kenneth D. A. Pillay  
Microprocessors & Microsystems archive  
Volume 14, Issue 7 (September 1990)  
*An example of a relocating loader for MS-DOS. Not the first one, but just a relatively modern example of how such a system works.*
- [SS74] "The Protection of Information in Computer Systems"  
J. Saltzer and M. Schroeder  
CACM, July 1974  
*From this paper: "The concepts of base-and-bound register and hardware-interpreted descriptors appeared, apparently independently, between 1957 and 1959 on three projects with diverse goals. At M.I.T., McCarthy suggested the base-and-bound idea as part of the memory protection system necessary to make time-sharing feasible. IBM independently developed the base-and-bound register as a mechanism to permit reliable multiprogramming of the Stretch (7030) computer system. At Burroughs, R. Barton suggested that hardware-interpreted descriptors would provide direct support for the naming scope rules of higher level languages in the B5000 computer system." We found this quote on Mark Smotherman's cool history pages [S04]; see them for more information.*
- [S04] "System Call Support"  
Mark Smotherman, May 2004  
<http://people.cs.clemson.edu/~mark/syscall.html>  
*A neat history of system call support. Smotherman has also collected some early history on items like interrupts and other fun aspects of computing history. See his web pages for more details.*
- [WL+93] "Efficient Software-based Fault Isolation"  
Robert Wahbe, Steven Lucco, Thomas E. Anderson, Susan L. Graham  
SOSP '93  
*A terrific paper about how you can use compiler support to bound memory references from a program, without hardware support. The paper sparked renewed interest in software techniques for isolation of memory references.*

## Homework

The program `relocation.py` allows you to see how address translations are performed in a system with base and bounds registers. See the README for details.

### Questions

1. Run with seeds 1, 2, and 3, and compute whether each virtual address generated by the process is in or out of bounds. If in bounds, compute the translation.
2. Run with these flags: `-s 0 -n 10`. What value do you have set `-l` (the bounds register) to in order to ensure that all the generated virtual addresses are within bounds?
3. Run with these flags: `-s 1 -n 10 -l 100`. What is the maximum value that bounds 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.

## Segmentation

So far we have been putting the entire address space of each process in memory. With the base and bounds registers, the OS can easily relocate processes to different parts of physical memory. However, you might have noticed something interesting about these address spaces of ours: there is a big chunk of “free” space right in the middle, between the stack and the heap.

As you can imagine from Figure 16.1, although the space between the stack and heap is not being used by the process, it is still taking up physical memory when we relocate the entire address space somewhere in physical memory; thus, the simple approach of using a base and bounds register pair to virtualize memory is wasteful. It also makes it quite hard to run a program when the entire address space doesn’t fit into memory; thus, base and bounds is not as flexible as we would like. And thus:

### THE CRUX: HOW TO SUPPORT A LARGE ADDRESS SPACE

How do we support a large address space with (potentially) a lot of free space between the stack and the heap? Note that in our examples, with tiny (pretend) address spaces, the waste doesn’t seem too bad. Imagine, however, a 32-bit address space (4 GB in size); a typical program will only use megabytes of memory, but still would demand that the entire address space be resident in memory.

### 16.1 Segmentation: Generalized Base/Bounds

To solve this problem, an idea was born, and it is called **segmentation**. It is quite an old idea, going at least as far back as the very early 1960’s [H61, G62]. The idea is simple: instead of having just one base and bounds pair in our MMU, why not have a base and bounds pair per logical **segment** of the address space? A segment is just a contiguous portion of the address space of a particular length, and in our canonical

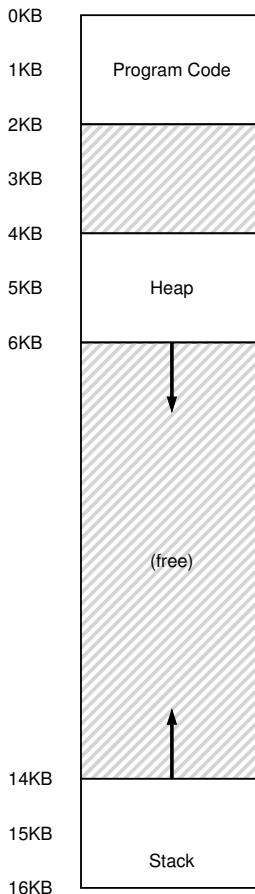


Figure 16.1: An Address Space (Again)

address space, we have three logically-different segments: code, stack, and heap. What segmentation allows the OS to do is to place each one of those segments in different parts of physical memory, and thus avoid filling physical memory with unused virtual address space.

Let's look at an example. Assume we want to place the address space from Figure 16.1 into physical memory. With a base and bounds pair per segment, we can place each segment *independently* in physical memory. For example, see Figure 16.2 (page 3); there you see a 64KB physical memory with those three segments in it (and 16KB reserved for the OS).

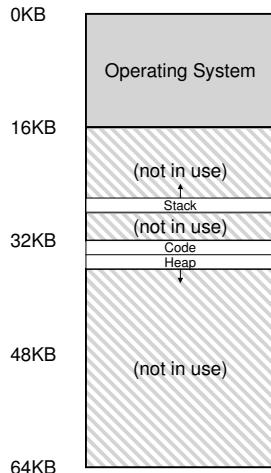


Figure 16.2: Placing Segments In Physical Memory

As you can see in the diagram, only used memory is allocated space in physical memory, and thus large address spaces with large amounts of unused address space (which we sometimes call **sparse address spaces**) can be accommodated.

The hardware structure in our MMU required to support segmentation is just what you'd expect: in this case, a set of three base and bounds register pairs. Figure 16.3 below shows the register values for the example above; each bounds register holds the size of a segment.

Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

Figure 16.3: Segment Register Values

You can see from the figure that the code segment is placed at physical address 32KB and has a size of 2KB and the heap segment is placed at 34KB and also has a size of 2KB.

Let's do an example translation, using the address space in Figure 16.1. Assume a reference is made to virtual address 100 (which is in the code segment). When the reference takes place (say, on an instruction fetch), the hardware will add the base value to the *offset* into this segment (100 in this case) to arrive at the desired physical address:  $100 + 32\text{KB}$ , or 32868. It will then check that the address is within bounds (100 is less than 2KB), find that it is, and issue the reference to physical memory address 32868.

**ASIDE: THE SEGMENTATION FAULT**

The term segmentation fault or violation arises from a memory access on a segmented machine to an illegal address. Humorously, the term persists, even on machines with no support for segmentation at all. Or not so humorously, if you can't figure why your code keeps faulting.

Now let's look at an address in the heap, virtual address 4200 (again refer to Figure 16.1). If we just add the virtual address 4200 to the base of the heap (34KB), we get a physical address of 39016, which is *not* the correct physical address. What we need to first do is extract the *offset* into the heap, i.e., which byte(s) *in this segment* the address refers to. Because the heap starts at virtual address 4KB (4096), the offset of 4200 is actually 4200 minus 4096, or 104. We then take this offset (104) and add it to the base register physical address (34K) to get the desired result: 34920.

What if we tried to refer to an illegal address, such as 7KB which is beyond the end of the heap? You can imagine what will happen: the hardware detects that the address is out of bounds, traps into the OS, likely leading to the termination of the offending process. And now you know the origin of the famous term that all C programmers learn to dread: the **segmentation violation** or **segmentation fault**.

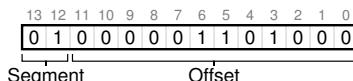
## 16.2 Which Segment Are We Referring To?

The hardware uses segment registers during translation. How does it know the offset into a segment, and to which segment an address refers?

One common approach, sometimes referred to as an **explicit** approach, is to chop up the address space into segments based on the top few bits of the virtual address; this technique was used in the VAX/VMS system [LL82]. In our example above, we have three segments; thus we need two bits to accomplish our task. If we use the top two bits of our 14-bit virtual address to select the segment, our virtual address looks like this:



In our example, then, if the top two bits are 00, the hardware knows the virtual address is in the code segment, and thus uses the code base and bounds pair to relocate the address to the correct physical location. If the top two bits are 01, the hardware knows the address is in the heap, and thus uses the heap base and bounds. Let's take our example heap virtual address from above (4200) and translate it, just to make sure this is clear. The virtual address 4200, in binary form, can be seen here:



As you can see from the picture, the top two bits (01) tell the hardware which *segment* we are referring to. The bottom 12 bits are the *offset* into the segment: 0000 0110 1000, or hex 0x068, or 104 in decimal. Thus, the hardware simply takes the first two bits to determine which segment register to use, and then takes the next 12 bits as the offset into the segment. By adding the base register to the offset, the hardware arrives at the final physical address. Note the offset eases the bounds check too: we can simply check if the offset is less than the bounds; if not, the address is illegal. Thus, if base and bounds were arrays (with one entry per segment), the hardware would be doing something like this to obtain the desired physical address:

```

1 // get top 2 bits of 14-bit VA
2 Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3 // now get offset
4 Offset  = VirtualAddress & OFFSET_MASK
5 if (Offset >= Bounds[Segment])
6     RaiseException(PROTECTION_FAULT)
7 else
8     PhysAddr = Base[Segment] + Offset
9     Register = AccessMemory(PhysAddr)
```

In our running example, we can fill in values for the constants above. Specifically, SEG\_MASK would be set to 0x3000, SEG\_SHIFT to 12, and OFFSET\_MASK to 0xFFFF.

You may also have noticed that when we use the top two bits, and we only have three segments (code, heap, stack), one segment of the address space goes unused. Thus, some systems put code in the same segment as the heap and thus use only one bit to select which segment to use [LL82].

There are other ways for the hardware to determine which segment a particular address is in. In the **implicit** approach, the hardware determines the segment by noticing how the address was formed. If, for example, the address was generated from the program counter (i.e., it was an instruction fetch), then the address is within the code segment; if the address is based off of the stack or base pointer, it must be in the stack segment; any other address must be in the heap.

### 16.3 What About The Stack?

Thus far, we've left out one important component of the address space: the stack. The stack has been relocated to physical address 28KB in the diagram above, but with one critical difference: *it grows backwards*. In physical memory, it starts at 28KB and grows back to 26KB, corresponding to virtual addresses 16KB to 14KB; translation must proceed differently.

The first thing we need is a little extra hardware support. Instead of just base and bounds values, the hardware also needs to know which way the segment grows (a bit, for example, that is set to 1 when the segment grows in the positive direction, and 0 for negative). Our updated view of what the hardware tracks is seen in Figure 16.4.

Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0

Figure 16.4: Segment Registers (With Negative-Growth Support)

With the hardware understanding that segments can grow in the negative direction, the hardware must now translate such virtual addresses slightly differently. Let's take an example stack virtual address and translate it to understand the process.

In this example, assume we wish to access virtual address 15KB, which should map to physical address 27KB. Our virtual address, in binary form, thus looks like this: 11 1100 0000 0000 (hex 0x3C00). The hardware uses the top two bits (11) to designate the segment, but then we are left with an offset of 3KB. To obtain the correct negative offset, we must subtract the maximum segment size from 3KB: in this example, a segment can be 4KB, and thus the correct negative offset is 3KB minus 4KB which equals -1KB. We simply add the negative offset (-1KB) to the base (28KB) to arrive at the correct physical address: 27KB. The bounds check can be calculated by ensuring the absolute value of the negative offset is less than the segment's size.

## 16.4 Support for Sharing

As support for segmentation grew, system designers soon realized that they could realize new types of efficiencies with a little more hardware support. Specifically, to save memory, sometimes it is useful to **share** certain memory segments between address spaces. In particular, **code sharing** is common and still in use in systems today.

To support sharing, we need a little extra support from the hardware, in the form of **protection bits**. Basic support adds a few bits per segment, indicating whether or not a program can read or write a segment, or perhaps execute code that lies within the segment. By setting a code segment to read-only, the same code can be shared across multiple processes, without worry of harming isolation; while each process still thinks that it is accessing its own private memory, the OS is secretly sharing memory which cannot be modified by the process, and thus the illusion is preserved.

An example of the additional information tracked by the hardware (and OS) is shown in Figure 16.5. As you can see, the code segment is set to read and execute, and thus the same physical segment in memory could be mapped into multiple virtual address spaces.

Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

Figure 16.5: Segment Register Values (with Protection)

With protection bits, the hardware algorithm described earlier would also have to change. In addition to checking whether a virtual address is within bounds, the hardware also has to check whether a particular access is permissible. If a user process tries to write to a read-only segment, or execute from a non-executable segment, the hardware should raise an exception, and thus let the OS deal with the offending process.

## 16.5 Fine-grained vs. Coarse-grained Segmentation

Most of our examples thus far have focused on systems with just a few segments (i.e., code, stack, heap); we can think of this segmentation as **coarse-grained**, as it chops up the address space into relatively large, coarse chunks. However, some early systems (e.g., Multics [CV65,DD68]) were more flexible and allowed for address spaces to consist of a large number of smaller segments, referred to as **fine-grained** segmentation.

Supporting many segments requires even further hardware support, with a **segment table** of some kind stored in memory. Such segment tables usually support the creation of a very large number of segments, and thus enable a system to use segments in more flexible ways than we have thus far discussed. For example, early machines like the Burroughs B5000 had support for thousands of segments, and expected a compiler to chop code and data into separate segments which the OS and hardware would then support [RK68]. The thinking at the time was that by having fine-grained segments, the OS could better learn about which segments are in use and which are not and thus utilize main memory more effectively.

## 16.6 OS Support

You now should have a basic idea as to how segmentation works. Pieces of the address space are relocated into physical memory as the system runs, and thus a huge savings of physical memory is achieved relative to our simpler approach with just a single base/bounds pair for the entire address space. Specifically, all the unused space between the stack and the heap need not be allocated in physical memory, allowing us to fit more address spaces into physical memory.

However, segmentation raises a number of new issues. We'll first describe the new OS issues that must be addressed. The first is an old one: what should the OS do on a context switch? You should have a good guess by now: the segment registers must be saved and restored. Clearly, each process has its own virtual address space, and the OS must make sure to set up these registers correctly before letting the process run again.

The second, and more important, issue is managing free space in physical memory. When a new address space is created, the OS has to be able to find space in physical memory for its segments. Previously, we assumed that each address space was the same size, and thus physical memory could be thought of as a bunch of slots where processes would fit in. Now, we have a number of segments per process, and each segment might be a different size.

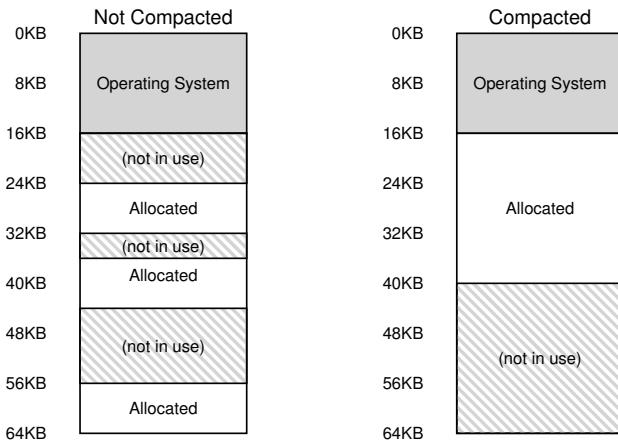


Figure 16.6: Non-compacted and Compacted Memory

The general problem that arises is that physical memory quickly becomes full of little holes of free space, making it difficult to allocate new segments, or to grow existing ones. We call this problem **external fragmentation** [R69]; see Figure 16.6 (left).

In the example, a process comes along and wishes to allocate a 20KB segment. In that example, there is 24KB free, but not in one contiguous segment (rather, in three non-contiguous chunks). Thus, the OS cannot satisfy the 20KB request.

One solution to this problem would be to **compact** physical memory by rearranging the existing segments. For example, the OS could stop whichever processes are running, copy their data to one contiguous region of memory, change their segment register values to point to the new physical locations, and thus have a large free extent of memory with which to work. By doing so, the OS enables the new allocation request to succeed. However, compaction is expensive, as copying segments is memory-intensive and generally uses a fair amount of processor time. See Figure 16.6 (right) for a diagram of compacted physical memory.

A simpler approach is to use a free-list management algorithm that tries to keep large extents of memory available for allocation. There are literally hundreds of approaches that people have taken, including classic algorithms like **best-fit** (which keeps a list of free spaces and returns the one closest in size that satisfies the desired allocation to the requester), **worst-fit**, **first-fit**, and more complex schemes like **buddy algorithm** [K68]. An excellent survey by Wilson et al. is a good place to start if you want to learn more about such algorithms [W+95], or you can wait until we cover some of the basics ourselves in a later chapter. Unfortunately, though, no matter how smart the algorithm, external fragmentation will still exist; thus, a good algorithm simply attempts to minimize it.

**TIP: IF 1000 SOLUTIONS EXIST, NO GREAT ONE DOES**

The fact that so many different algorithms exist to try to minimize external fragmentation is indicative of a stronger underlying truth: there is no one “best” way to solve the problem. Thus, we settle for something reasonable and hope it is good enough. The only real solution (as we will see in forthcoming chapters) is to avoid the problem altogether, by never allocating memory in variable-sized chunks.

## 16.7 Summary

Segmentation solves a number of problems, and helps us build a more effective virtualization of memory. Beyond just dynamic relocation, segmentation can better support sparse address spaces, by avoiding the huge potential waste of memory between logical segments of the address space. It is also fast, as doing the arithmetic segmentation requires is easy and well-suited to hardware; the overheads of translation are minimal. A fringe benefit arises too: code sharing. If code is placed within a separate segment, such a segment could potentially be shared across multiple running programs.

However, as we learned, allocating variable-sized segments in memory leads to some problems that we’d like to overcome. The first, as discussed above, is external fragmentation. Because segments are variable-sized, free memory gets chopped up into odd-sized pieces, and thus satisfying a memory-allocation request can be difficult. One can try to use smart algorithms [W+95] or periodically compact memory, but the problem is fundamental and hard to avoid.

The second and perhaps more important problem is that segmentation still isn’t flexible enough to support our fully generalized, sparse address space. For example, if we have a large but sparsely-used heap all in one logical segment, the entire heap must still reside in memory in order to be accessed. In other words, if our model of how the address space is being used doesn’t exactly match how the underlying segmentation has been designed to support it, segmentation doesn’t work very well. We thus need to find some new solutions. Ready to find them?

## References

- [CV65] "Introduction and Overview of the Multics System"  
F. J. Corbato and V. A. Vyssotsky  
Fall Joint Computer Conference, 1965  
*One of five papers presented on Multics at the Fall Joint Computer Conference; oh to be a fly on the wall in that room that day!*
- [DD68] "Virtual Memory, Processes, and Sharing in Multics"  
Robert C. Daley and Jack B. Dennis  
Communications of the ACM, Volume 11, Issue 5, May 1968  
*An early paper on how to perform dynamic linking in Multics, which was way ahead of its time. Dynamic linking finally found its way back into systems about 20 years later, as the large X-windows libraries demanded it. Some say that these large X11 libraries were MIT's revenge for removing support for dynamic linking in early versions of UNIX!*
- [G62] "Fact Segmentation"  
M. N. Greenfield  
Proceedings of the SJCC, Volume 21, May 1962  
*Another early paper on segmentation; so early that it has no references to other work.*
- [H61] "Program Organization and Record Keeping for Dynamic Storage"  
A. W. Holt  
Communications of the ACM, Volume 4, Issue 10, October 1961  
*An incredibly early and difficult to read paper about segmentation and some of its uses.*
- [I09] "Intel 64 and IA-32 Architectures Software Developer's Manuals"  
Intel, 2009  
Available: <http://www.intel.com/products/processor/manuals>  
*Try reading about segmentation in here (Chapter 3 in Volume 3a); it'll hurt your head, at least a little bit.*
- [K68] "The Art of Computer Programming: Volume I"  
Donald Knuth  
Addison-Wesley, 1968  
*Knuth is famous not only for his early books on the Art of Computer Programming but for his typesetting system TeX which is still a powerhouse typesetting tool used by professionals today, and indeed to typeset this very book. His tomes on algorithms are a great early reference to many of the algorithms that underly computing systems today.*
- [L83] "Hints for Computer Systems Design"  
Butler Lampson  
ACM Operating Systems Review, 15:5, October 1983  
*A treasure-trove of sage advice on how to build systems. Hard to read in one sitting; take it in a little at a time, like a fine wine, or a reference manual.*
- [LL82] "Virtual Memory Management in the VAX/VMS Operating System"  
Henry M. Levy and Peter H. Lipman  
IEEE Computer, Volume 15, Number 3 (March 1982)  
*A classic memory management system, with lots of common sense in its design. We'll study it in more detail in a later chapter.*

[RK68] "Dynamic Storage Allocation Systems"

B. Randell and C.J. Kuehner

Communications of the ACM

Volume 11(5), pages 297-306, May 1968

*A nice overview of the differences between paging and segmentation, with some historical discussion of various machines.*

[R69] "A note on storage fragmentation and program segmentation"

Brian Randell

Communications of the ACM

Volume 12(7), pages 365-372, July 1969

*One of the earliest papers to discuss fragmentation.*

[W+95] "Dynamic Storage Allocation: A Survey and Critical Review"

Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles

In International Workshop on Memory Management

Scotland, United Kingdom, September 1995

*A great survey paper on memory allocators.*

## Homework

This program allows you to see how address translations are performed in a system with segmentation. See the README for details.

### Questions

1. First let's use a tiny address space to translate some addresses. Here's a simple set of parameters with a few different random seeds; can you translate the addresses?

```
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0  
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1  
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2
```

2. Now, let's see if we understand this tiny address space we've constructed (using the parameters from the question above). What is the highest legal virtual address in segment 0? What about the lowest legal virtual address in segment 1? What are the lowest and highest *illegal* addresses in this entire address space? Finally, how would you run `segmentation.py` with the `-A` flag to test if you are right?
3. Let's say we have a tiny 16-byte address space in a 128-byte physical memory. What base and bounds would you set up so as to get the simulator to generate the following translation results for the specified address stream: valid, valid, violation, ..., violation, valid, valid? Assume the following parameters:

```
segmentation.py -a 16 -p 128  
-A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15  
--b0 ? --b1 ? --l1 ?
```

4. Assuming we want to generate a problem where roughly 90% of the randomly-generated virtual addresses are valid (i.e., not segmentation violations). How should you configure the simulator to do so? Which parameters are important?
5. Can you run the simulator such that no virtual addresses are valid? How?

## Free-Space Management

In this chapter, we take a small detour from our discussion of virtualizing memory to discuss a fundamental aspect of any memory management system, whether it be a malloc library (managing pages of a process's heap) or the OS itself (managing portions of the address space of a process). Specifically, we will discuss the issues surrounding **free-space management**.

Let us make the problem more specific. Managing free space can certainly be easy, as we will see when we discuss the concept of **paging**. It is easy when the space you are managing is divided into fixed-sized units; in such a case, you just keep a list of these fixed-sized units; when a client requests one of them, return the first entry.

Where free-space management becomes more difficult (and interesting) is when the free space you are managing consists of variable-sized units; this arises in a user-level memory-allocation library (as in `malloc()` and `free()`) and in an OS managing physical memory when using **segmentation** to implement virtual memory. In either case, the problem that exists is known as **external fragmentation**: the free space gets chopped into little pieces of different sizes and is thus fragmented; subsequent requests may fail because there is no single contiguous space that can satisfy the request, even though the total amount of free space exceeds the size of the request.



The figure shows an example of this problem. In this case, the total free space available is 20 bytes; unfortunately, it is fragmented into two chunks of size 10 each. As a result, a request for 15 bytes will fail even though there are 20 bytes free. And thus we arrive at the problem addressed in this chapter.

### CRUX: HOW TO MANAGE FREE SPACE

How should free space be managed, when satisfying variable-sized requests? What strategies can be used to minimize fragmentation? What are the time and space overheads of alternate approaches?

## 17.1 Assumptions

Most of this discussion will focus on the great history of allocators found in user-level memory-allocation libraries. We draw on Wilson's excellent survey [W+95] but encourage interested readers to go to the source document itself for more details<sup>1</sup>.

We assume a basic interface such as that provided by `malloc()` and `free()`. Specifically, `void *malloc(size_t size)` takes a single parameter, `size`, which is the number of bytes requested by the application; it hands back a pointer (of no particular type, or a **void pointer** in C lingo) to a region of that size (or greater). The complementary routine `void free(void *ptr)` takes a pointer and frees the corresponding chunk. Note the implication of the interface: the user, when freeing the space, does not inform the library of its size; thus, the library must be able to figure out how big a chunk of memory is when handed just a pointer to it. We'll discuss how to do this a bit later on in the chapter.

The space that this library manages is known historically as the **heap**, and the generic data structure used to manage free space in the heap is some kind of **free list**. This structure contains references to all of the free chunks of space in the managed region of memory. Of course, this data structure need not be a list *per se*, but just some kind of data structure to track free space.

We further assume that primarily we are concerned with **external fragmentation**, as described above. Allocators could of course also have the problem of **internal fragmentation**; if an allocator hands out chunks of memory bigger than that requested, any unasked for (and thus unused) space in such a chunk is considered *internal* fragmentation (because the waste occurs inside the allocated unit) and is another example of space waste. However, for the sake of simplicity, and because it is the more interesting of the two types of fragmentation, we'll mostly focus on external fragmentation.

We'll also assume that once memory is handed out to a client, it cannot be relocated to another location in memory. For example, if a program calls `malloc()` and is given a pointer to some space within the heap, that memory region is essentially "owned" by the program (and cannot be moved by the library) until the program returns it via a corresponding call to `free()`. Thus, no **compaction** of free space is possible, which

---

<sup>1</sup>It is nearly 80 pages long; thus, you really have to be interested!

would be useful to combat fragmentation<sup>2</sup>. Compaction could, however, be used in the OS to deal with fragmentation when implementing **segmentation** (as discussed in said chapter on segmentation).

Finally, we'll assume that the allocator manages a contiguous region of bytes. In some cases, an allocator could ask for that region to grow; for example, a user-level memory-allocation library might call into the kernel to grow the heap (via a system call such as `sbrk`) when it runs out of space. However, for simplicity, we'll just assume that the region is a single fixed size throughout its life.

## 17.2 Low-level Mechanisms

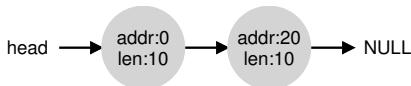
Before delving into some policy details, we'll first cover some common mechanisms used in most allocators. First, we'll discuss the basics of splitting and coalescing, common techniques in most any allocator. Second, we'll show how one can track the size of allocated regions quickly and with relative ease. Finally, we'll discuss how to build a simple list inside the free space to keep track of what is free and what isn't.

### Splitting and Coalescing

A free list contains a set of elements that describe the free space still remaining in the heap. Thus, assume the following 30-byte heap:

	free	used	free	
0	10	20	30	

The free list for this heap would have two elements on it. One entry describes the first 10-byte free segment (bytes 0-9), and one entry describes the other free segment (bytes 20-29):



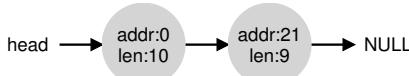
As described above, a request for anything greater than 10 bytes will fail (returning `NULL`); there just isn't a single contiguous chunk of memory of that size available. A request for exactly that size (10 bytes) could be satisfied easily by either of the free chunks. But what happens if the request is for something *smaller* than 10 bytes?

Assume we have a request for just a single byte of memory. In this case, the allocator will perform an action known as **splitting**: it will find

---

<sup>2</sup>Once you hand a pointer to a chunk of memory to a C program, it is generally difficult to determine all references (pointers) to that region, which may be stored in other variables or even in registers at a given point in execution. This may not be the case in more strongly-typed, garbage-collected languages, which would thus enable compaction as a technique to combat fragmentation.

a free chunk of memory that can satisfy the request and split it into two. The first chunk it will return to the caller; the second chunk will remain on the list. Thus, in our example above, if a request for 1 byte were made, and the allocator decided to use the second of the two elements on the list to satisfy the request, the call to `malloc()` would return 20 (the address of the 1-byte allocated region) and the list would end up looking like this:



In the picture, you can see the list basically stays intact; the only change is that the free region now starts at 21 instead of 20, and the length of that free region is now just 9<sup>3</sup>. Thus, the split is commonly used in allocators when requests are smaller than the size of any particular free chunk.

A corollary mechanism found in many allocators is known as **coalescing** of free space. Take our example from above once more (free 10 bytes, used 10 bytes, and another free 10 bytes).

Given this (tiny) heap, what happens when an application calls `free(10)`, thus returning the space in the middle of the heap? If we simply add this free space back into our list without too much thinking, we might end up with a list that looks like this:



Note the problem: while the entire heap is now free, it is seemingly divided into three chunks of 10 bytes each. Thus, if a user requests 20 bytes, a simple list traversal will not find such a free chunk, and return failure.

What allocators do in order to avoid this problem is coalesce free space when a chunk of memory is freed. The idea is simple: when returning a free chunk in memory, look carefully at the addresses of the chunk you are returning as well as the nearby chunks of free space; if the newly-free space sits right next to one (or two, as in this example) existing free chunks, merge them into a single larger free chunk. Thus, with coalescing, our final list should look like this:



Indeed, this is what the heap list looked like at first, before any allocations were made. With coalescing, an allocator can better ensure that large free extents are available for the application.

---

<sup>3</sup>This discussion assumes that there are no headers, an unrealistic but simplifying assumption we make for now.

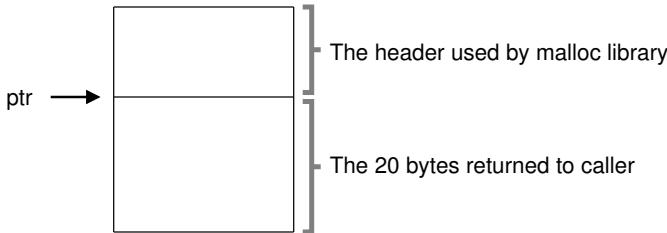


Figure 17.1: An Allocated Region Plus Header

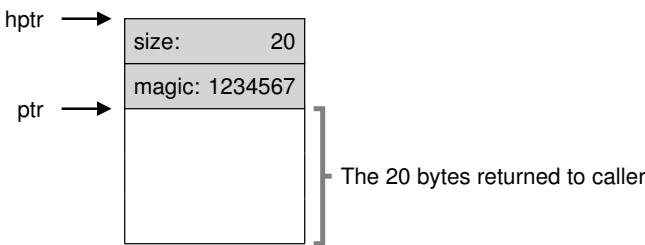


Figure 17.2: Specific Contents Of The Header

## Tracking The Size Of Allocated Regions

You might have noticed that the interface to `free(void *ptr)` does not take a size parameter; thus it is assumed that given a pointer, the malloc library can quickly determine the size of the region of memory being freed and thus incorporate the space back into the free list.

To accomplish this task, most allocators store a little bit of extra information in a **header** block which is kept in memory, usually just before the handed-out chunk of memory. Let's look at an example again (Figure 17.1). In this example, we are examining an allocated block of size 20 bytes, pointed to by `ptr`; imagine the user called `malloc()` and stored the results in `ptr`, e.g., `ptr = malloc(20);`.

The header minimally contains the size of the allocated region (in this case, 20); it may also contain additional pointers to speed up deallocation, a magic number to provide additional integrity checking, and other information. Let's assume a simple header which contains the size of the region and a magic number, like this:

```
typedef struct __header_t {
    int size;
    int magic;
} header_t;
```

The example above would look like what you see in Figure 17.2. When

the user calls `free(ptr)`, the library then uses simple pointer arithmetic to figure out where the header begins:

```
void free(void *ptr) {
    header_t *hptr = (void *)ptr - sizeof(header_t);
    ...
}
```

After obtaining such a pointer to the header, the library can easily determine whether the magic number matches the expected value as a sanity check (`assert(hptr->magic == 1234567)`) and calculate the total size of the newly-freed region via simple math (i.e., adding the size of the header to size of the region). Note the small but critical detail in the last sentence: the size of the free region is the size of the header plus the size of the space allocated to the user. Thus, when a user requests  $N$  bytes of memory, the library does not search for a free chunk of size  $N$ ; rather, it searches for a free chunk of size  $N$  plus the size of the header.

## Embedding A Free List

Thus far we have treated our simple free list as a conceptual entity; it is just a list describing the free chunks of memory in the heap. But how do we build such a list inside the free space itself?

In a more typical list, when allocating a new node, you would just call `malloc()` when you need space for the node. Unfortunately, within the memory-allocation library, you can't do this! Instead, you need to build the list *inside* the free space itself. Don't worry if this sounds a little weird; it is, but not so weird that you can't do it!

Assume we have a 4096-byte chunk of memory to manage (i.e., the heap is 4KB). To manage this as a free list, we first have to initialize said list; initially, the list should have one entry, of size 4096 (minus the header size). Here is the description of a node of the list:

```
typedef struct __node_t {
    int           size;
    struct __node_t *next;
} node_t;
```

Now let's look at some code that initializes the heap and puts the first element of the free list inside that space. We are assuming that the heap is built within some free space acquired via a call to the system call `mmap()`; this is not the only way to build such a heap but serves us well in this example. Here is the code:

```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                    MAP_ANON|MAP_PRIVATE, -1, 0);
head->size   = 4096 - sizeof(node_t);
head->next   = NULL;
```

After running this code, the status of the list is that it has a single entry, of size 4088. Yes, this is a tiny heap, but it serves as a fine example for us

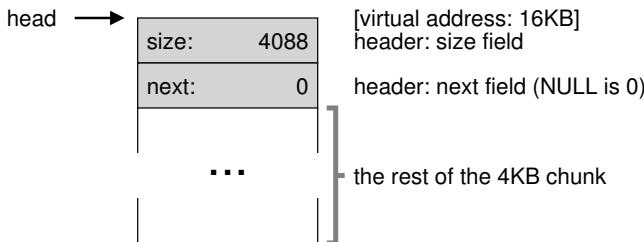


Figure 17.3: A Heap With One Free Chunk

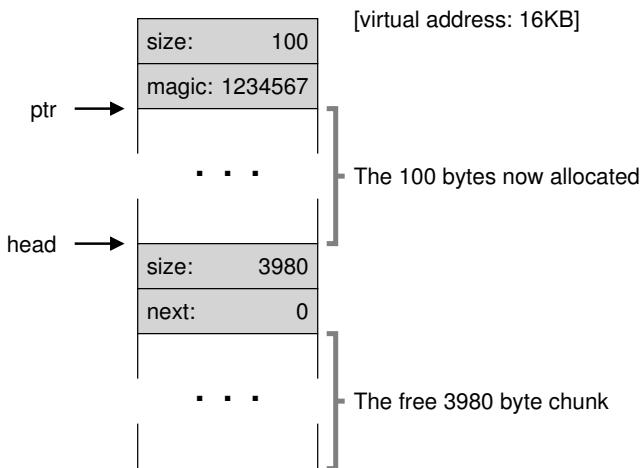


Figure 17.4: A Heap: After One Allocation

here. The `head` pointer contains the beginning address of this range; let's assume it is 16KB (though any virtual address would be fine). Visually, the heap thus looks like what you see in Figure 17.3.

Now, let's imagine that a chunk of memory is requested, say of size 100 bytes. To service this request, the library will first find a chunk that is large enough to accommodate the request; because there is only one free chunk (`size: 4088`), this chunk will be chosen. Then, the chunk will be **split** into two: one chunk big enough to service the request (and header, as described above), and the remaining free chunk. Assuming an 8-byte header (an integer size and an integer magic number), the space in the heap now looks like what you see in Figure 17.4.

Thus, upon the request for 100 bytes, the library allocated 108 bytes out of the existing one free chunk, returns a pointer (marked `ptr` in the figure above) to it, stashes the header information immediately before the

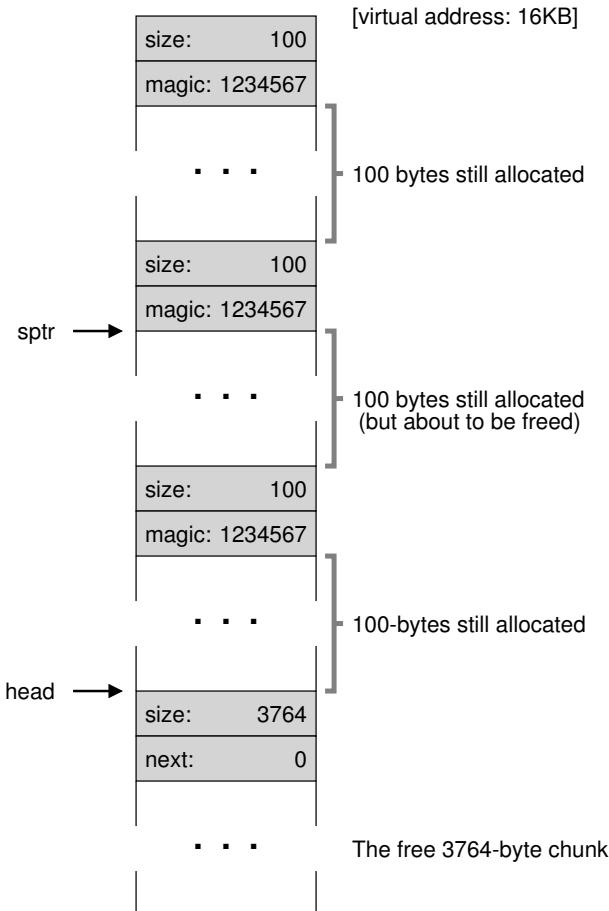


Figure 17.5: Free Space With Three Chunks Allocated

allocated space for later use upon `free()`, and shrinks the one free node in the list to 3980 bytes (4088 minus 108).

Now let's look at the heap when there are three allocated regions, each of 100 bytes (or 108 including the header). A visualization of this heap is shown in Figure 17.5.

As you can see therein, the first 324 bytes of the heap are now allocated, and thus we see three headers in that space as well as three 100-byte regions being used by the calling program. The free list remains uninteresting: just a single node (pointed to by `head`), but now only 3764 bytes in size after the three splits. But what happens when the calling program returns some memory via `free()`?

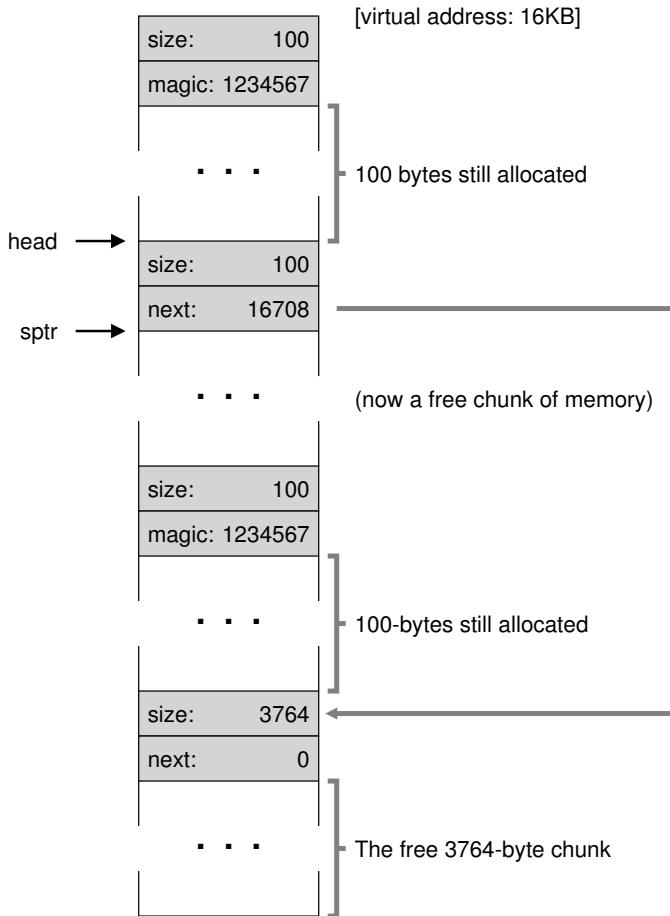


Figure 17.6: Free Space With Two Chunks Allocated

In this example, the application returns the middle chunk of allocated memory, by calling `free(16500)` (the value 16500 is arrived upon by adding the start of the memory region, 16384, to the 108 of the previous chunk and the 8 bytes of the header for this chunk). This value is shown in the previous diagram by the pointer `sprt`.

The library immediately figures out the size of the free region, and then adds the free chunk back onto the free list. Assuming we insert at the head of the free list, the space now looks like this (Figure 17.6).

And now we have a list that starts with a small free chunk (100 bytes, pointed to by the head of the list) and a large free chunk (3764 bytes).

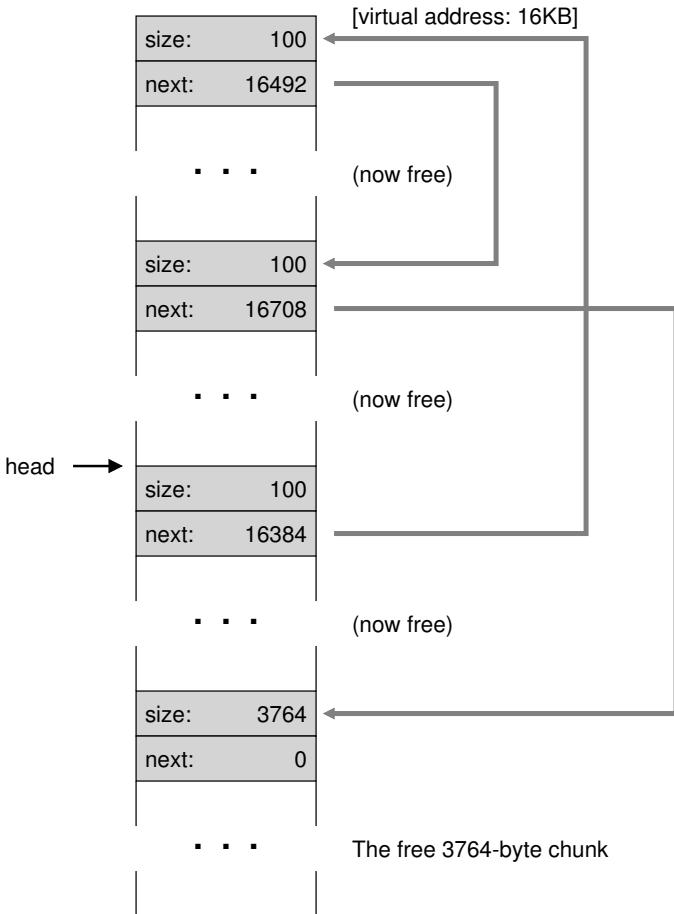


Figure 17.7: A Non-Coalesced Free List

Our list finally has more than one element on it! And yes, the free space is fragmented, an unfortunate but common occurrence.

One last example: let's assume now that the last two in-use chunks are freed. Without coalescing, you might end up with a free list that is highly fragmented (see Figure 17.7).

As you can see from the figure, we now have a big mess! Why? Simple, we forgot to **coalesce** the list. Although all of the memory is free, it is chopped up into pieces, thus appearing as a fragmented memory despite not being one. The solution is simple: go through the list and **merge** neighboring chunks; when finished, the heap will be whole again.

## Growing The Heap

We should discuss one last mechanism found within many allocation libraries. Specifically, what should you do if the heap runs out of space? The simplest approach is just to fail. In some cases this is the only option, and thus returning NULL is an honorable approach. Don't feel bad! You tried, and though you failed, you fought the good fight.

Most traditional allocators start with a small-sized heap and then request more memory from the OS when they run out. Typically, this means they make some kind of system call (e.g., `sbrk` in most UNIX systems) to grow the heap, and then allocate the new chunks from there. To service the `sbrk` request, the OS finds free physical pages, maps them into the address space of the requesting process, and then returns the value of the end of the new heap; at that point, a larger heap is available, and the request can be successfully serviced.

### 17.3 Basic Strategies

Now that we have some machinery under our belt, let's go over some basic strategies for managing free space. These approaches are mostly based on pretty simple policies that you could think up yourself; try it before reading and see if you come up with all of the alternatives (or maybe some new ones!).

The ideal allocator is both fast and minimizes fragmentation. Unfortunately, because the stream of allocation and free requests can be arbitrary (after all, they are determined by the programmer), any particular strategy can do quite badly given the wrong set of inputs. Thus, we will not describe a "best" approach, but rather talk about some basics and discuss their pros and cons.

#### Best Fit

The **best fit** strategy is quite simple: first, search through the free list and find chunks of free memory that are as big or bigger than the requested size. Then, return the one that is the smallest in that group of candidates; this is the so called best-fit chunk (it could be called smallest fit too). One pass through the free list is enough to find the correct block to return.

The intuition behind best fit is simple: by returning a block that is close to what the user asks, best fit tries to reduce wasted space. However, there is a cost; naive implementations pay a heavy performance penalty when performing an exhaustive search for the correct free block.

#### Worst Fit

The **worst fit** approach is the opposite of best fit; find the largest chunk and return the requested amount; keep the remaining (large) chunk on the free list. Worst fit tries to thus leave big chunks free instead of lots of

small chunks that can arise from a best-fit approach. Once again, however, a full search of free space is required, and thus this approach can be costly. Worse, most studies show that it performs badly, leading to excess fragmentation while still having high overheads.

### First Fit

The **first fit** method simply finds the first block that is big enough and returns the requested amount to the user. As before, the remaining free space is kept free for subsequent requests.

First fit has the advantage of speed — no exhaustive search of all the free spaces are necessary — but sometimes pollutes the beginning of the free list with small objects. Thus, how the allocator manages the free list's order becomes an issue. One approach is to use **address-based ordering**; by keeping the list ordered by the address of the free space, coalescing becomes easier, and fragmentation tends to be reduced.

### Next Fit

Instead of always beginning the first-fit search at the beginning of the list, the **next fit** algorithm keeps an extra pointer to the location within the list where one was looking last. The idea is to spread the searches for free space throughout the list more uniformly, thus avoiding splintering of the beginning of the list. The performance of such an approach is quite similar to first fit, as an exhaustive search is once again avoided.

### Examples

Here are a few examples of the above strategies. Envision a free list with three elements on it, of sizes 10, 30, and 20 (we'll ignore headers and other details here, instead just focusing on how strategies operate):



Assume an allocation request of size 15. A best-fit approach would search the entire list and find that 20 was the best fit, as it is the smallest free space that can accommodate the request. The resulting free list:



As happens in this example, and often happens with a best-fit approach, a small free chunk is now left over. A worst-fit approach is similar but instead finds the largest chunk, in this example 30. The resulting list:



The first-fit strategy, in this example, does the same thing as worst-fit, also finding the first free block that can satisfy the request. The difference is in the search cost; both best-fit and worst-fit look through the entire list; first-fit only examines free chunks until it finds one that fits, thus reducing search cost.

These examples just scratch the surface of allocation policies. More detailed analysis with real workloads and more complex allocator behaviors (e.g., coalescing) are required for a deeper understanding. Perhaps something for a homework section, you say?

## 17.4 Other Approaches

Beyond the basic approaches described above, there have been a host of suggested techniques and algorithms to improve memory allocation in some way. We list a few of them here for your consideration (i.e., to make you think about a little more than just best-fit allocation).

### Segregated Lists

One interesting approach that has been around for some time is the use of **segregated lists**. The basic idea is simple: if a particular application has one (or a few) popular-sized request that it makes, keep a separate list just to manage objects of that size; all other requests are forwarded to a more general memory allocator.

The benefits of such an approach are obvious. By having a chunk of memory dedicated for one particular size of requests, fragmentation is much less of a concern; moreover, allocation and free requests can be served quite quickly when they are of the right size, as no complicated search of a list is required.

Just like any good idea, this approach introduces new complications into a system as well. For example, how much memory should one dedicate to the pool of memory that serves specialized requests of a given size, as opposed to the general pool? One particular allocator, the **slab allocator** by uber-engineer Jeff Bonwick (which was designed for use in the Solaris kernel), handles this issue in a rather nice way [B94].

Specifically, when the kernel boots up, it allocates a number of **object caches** for kernel objects that are likely to be requested frequently (such as locks, file-system inodes, etc.); the object caches thus are each segregated free lists of a given size and serve memory allocation and free requests quickly. When a given cache is running low on free space, it requests some **slabs** of memory from a more general memory allocator (the total amount requested being a multiple of the page size and the object in question). Conversely, when the reference counts of the objects within a given slab all go to zero, the general allocator can reclaim them from the specialized allocator, which is often done when the VM system needs more memory.

**ASIDE: GREAT ENGINEERS ARE REALLY GREAT**

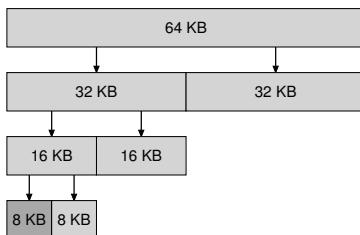
Engineers like Jeff Bonwick (who not only wrote the slab allocator mentioned herein but also was the lead of an amazing file system, ZFS) are the heart of Silicon Valley. Behind almost any great product or technology is a human (or small group of humans) who are way above average in their talents, abilities, and dedication. As Mark Zuckerberg (of Facebook) says: “Someone who is exceptional in their role is not just a little better than someone who is pretty good. They are 100 times better.” This is why, still today, one or two people can start a company that changes the face of the world forever (think Google, Apple, or Facebook). Work hard and you might become such a “100x” person as well. Failing that, work *with* such a person; you’ll learn more in a day than most learn in a month. Failing that, feel sad.

The slab allocator also goes beyond most segregated list approaches by keeping free objects on the lists in a pre-initialized state. Bonwick shows that initialization and destruction of data structures is costly [B94]; by keeping freed objects in a particular list in their initialized state, the slab allocator thus avoids frequent initialization and destruction cycles per object and thus lowers overheads noticeably.

## Buddy Allocation

Because coalescing is critical for an allocator, some approaches have been designed around making coalescing simple. One good example is found in the **binary buddy allocator** [K65].

In such a system, free memory is first conceptually thought of as one big space of size  $2^N$ . When a request for memory is made, the search for free space recursively divides free space by two until a block that is big enough to accommodate the request is found (and a further split into two would result in a space that is too small). At this point, the requested block is returned to the user. Here is an example of a 64KB free space getting divided in the search for a 7KB block:



In the example, the leftmost 8KB block is allocated (as indicated by the darker shade of gray) and returned to the user; note that this scheme can suffer from **internal fragmentation**, as you are only allowed to give out power-of-two-sized blocks.

The beauty of buddy allocation is found in what happens when that block is freed. When returning the 8KB block to the free list, the allocator checks whether the “buddy” 8KB is free; if so, it coalesces the two blocks into a 16KB block. The allocator then checks if the buddy of the 16KB block is still free; if so, it coalesces those two blocks. This recursive coalescing process continues up the tree, either restoring the entire free space or stopping when a buddy is found to be in use.

The reason buddy allocation works so well is that it is simple to determine the buddy of a particular block. How, you ask? Think about the addresses of the blocks in the free space above. If you think carefully enough, you’ll see that the address of each buddy pair only differs by a single bit; which bit is determined by the level in the buddy tree. And thus you have a basic idea of how binary buddy allocation schemes work. For more detail, as always, see the Wilson survey [W+95].

### Other Ideas

One major problem with many of the approaches described above is their lack of **scaling**. Specifically, searching lists can be quite slow. Thus, advanced allocators use more complex data structures to address these costs, trading simplicity for performance. Examples include balanced binary trees, splay trees, or partially-ordered trees [W+95].

Given that modern systems often have multiple processors and run multi-threaded workloads (something you’ll learn about in great detail in the section of the book on Concurrency), it is not surprising that a lot of effort has been spent making allocators work well on multiprocessor-based systems. Two wonderful examples are found in Berger et al. [B+00] and Evans [E06]; check them out for the details.

These are but two of the thousands of ideas people have had over time about memory allocators; read on your own if you are curious. Failing that, read about how the glibc allocator works [S15], to give you a sense of what the real world is like.

## 17.5 Summary

In this chapter, we’ve discussed the most rudimentary forms of memory allocators. Such allocators exist everywhere, linked into every C program you write, as well as in the underlying OS which is managing memory for its own data structures. As with many systems, there are many trade-offs to be made in building such a system, and the more you know about the exact workload presented to an allocator, the more you could do to tune it to work better for that workload. Making a fast, space-efficient, scalable allocator that works well for a broad range of workloads remains an on-going challenge in modern computer systems.

## References

- [B+00] "Hoard: A Scalable Memory Allocator for Multithreaded Applications"  
 Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson  
 ASPLOS-IX, November 2000  
*Berger and company's excellent allocator for multiprocessor systems. Beyond just being a fun paper, also used in practice!*
- [B94] "The Slab Allocator: An Object-Caching Kernel Memory Allocator"  
 Jeff Bonwick  
 USENIX '94  
*A cool paper about how to build an allocator for an operating system kernel, and a great example of how to specialize for particular common object sizes.*
- [E06] "A Scalable Concurrent malloc(3) Implementation for FreeBSD"  
 Jason Evans  
<http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf>  
 April 2006  
*A detailed look at how to build a real modern allocator for use in multiprocessors. The "jemalloc" allocator is in widespread use today, within FreeBSD, NetBSD, Mozilla Firefox, and within Facebook.*
- [K65] "A Fast Storage Allocator"  
 Kenneth C. Knowlton  
 Communications of the ACM, Volume 8, Number 10, October 1965  
*The common reference for buddy allocation. Random strange fact: Knuth gives credit for the idea not to Knowlton but to Harry Markowitz, a Nobel-prize winning economist. Another strange fact: Knuth communicates all of his emails via a secretary; he doesn't send email himself, rather he tells his secretary what email to send and then the secretary does the work of emailing. Last Knuth fact: he created TeX, the tool used to typeset this book. It is an amazing piece of software<sup>4</sup>.*
- [S15] "Understanding glibc malloc"  
 Sploitfun  
 February, 2015  
<https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/>  
*A deep dive into how glibc malloc works. Amazingly detailed and a very cool read.*
- [W+95] "Dynamic Storage Allocation: A Survey and Critical Review"  
 Paul R. Wilson, Mark S. Johnstone, Michael Neely, David Boles  
 International Workshop on Memory Management  
 Kinross, Scotland, September 1995  
*An excellent and far-reaching survey of many facets of memory allocation. Far too much detail to go into in this tiny chapter!*

---

<sup>4</sup> Actually we use LaTeX, which is based on Lamport's additions to TeX, but close enough.

## Homework

The program, `malloc.py`, lets you explore the behavior of a simple free-space allocator as described in the chapter. See the README for details of its basic operation.

## Questions

1. First run with the flags `-n 10 -H 0 -p BEST -s 0` to generate a few random allocations and frees. Can you predict what `alloc()/free()` will return? Can you guess the state of the free list after each request? What do you notice about the free list over time?
2. How are the results different when using a WORST fit policy to search the free list (`-p WORST`)? What changes?
3. What about when using FIRST fit (`-p FIRST`)? What speeds up when you use first fit?
4. For the above questions, how the list is kept ordered can affect the time it takes to find a free location for some of the policies. Use the different free list orderings (`-l ADDRSORT, -l SIZESORT+, -l SIZESORT-`) to see how the policies and the list orderings interact.
5. Coalescing of a free list can be quite important. Increase the number of random allocations (say to `-n 1000`). What happens to larger allocation requests over time? Run with and without coalescing (i.e., without and with the `-c` flag). What differences in outcome do you see? How big is the free list over time in each case? Does the ordering of the list matter in this case?
6. What happens when you change the percent allocated fraction `-p` to higher than 50? What happens to allocations as it nears 100? What about as it nears 0?
7. What kind of specific requests can you make to generate a highly-fragmented free space? Use the `-A` flag to create fragmented free lists, and see how different policies and options change the organization of the free list.

## 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<sup>1</sup>; in the book, we'll often use tiny examples to make them easier to digest.

---

<sup>1</sup>A 64-bit address space is hard to imagine, it is so amazingly large. An analogy might help: if you think of a 32-bit address space as the size of a tennis court, a 64-bit address space is about the size of Europe(!).

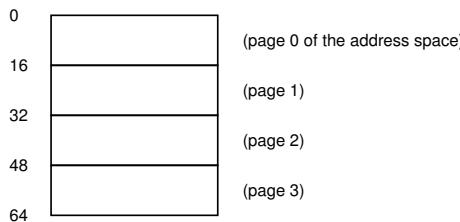


Figure 18.1: A Simple 64-byte Address Space

Physical memory, as shown in Figure 18.2, also consists of a number of fixed-sized slots, in this case eight page frames (making for a 128-byte physical memory, also ridiculously small). As you can see in the diagram, the pages of the virtual address space have been placed at different locations throughout physical memory; the diagram also shows the OS using some of physical memory for itself.

Paging, as we will see, has a number of advantages over our previous approaches. Probably the most important improvement will be *flexibility*: with a fully-developed paging approach, the system will be able to support the abstraction of an address space effectively, regardless of how a process uses the address space; we won't, for example, make assumptions about the direction the heap and stack grow and how they are used.

Another advantage is the *simplicity* of free-space management that paging affords. For example, when the OS wishes to place our tiny 64-byte address space into our eight-page physical memory, it simply finds four free pages; perhaps the OS keeps a **free list** of all free pages for this, and just grabs the first four free pages off of this list. In the example, the OS

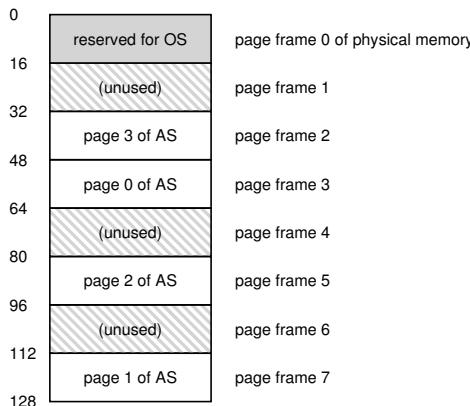


Figure 18.2: A 64-Byte Address Space In A 128-Byte Physical Memory

has placed virtual page 0 of the address space (AS) in physical frame 3, virtual page 1 of the AS in physical frame 7, page 2 in frame 5, and page 3 in frame 2. Page frames 1, 4, and 6 are currently free.

To record where each virtual page of the address space is placed in physical memory, the operating system usually keeps a *per-process* data structure known as a **page table**. The major role of the page table is to store **address translations** for each of the virtual pages of the address space, thus letting us know where in physical memory each page resides. For our simple example (Figure 18.2, page 2), the page table would thus have the following four entries: (Virtual Page 0 → Physical Frame 3), (VP 1 → PF 7), (VP 2 → PF 5), and (VP 3 → PF 2).

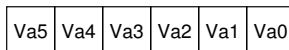
It is important to remember that this page table is a *per-process* data structure (most page table structures we discuss are per-process structures; an exception we'll touch on is the **inverted page table**). If another process were to run in our example above, the OS would have to manage a different page table for it, as its virtual pages obviously map to *different* physical pages (modulo any sharing going on).

Now, we know enough to perform an address-translation example. Let's imagine the process with that tiny address space (64 bytes) is performing a memory access:

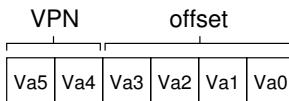
```
movl <virtual address>, %eax
```

Specifically, let's pay attention to the explicit load of the data from address <virtual address> into the register `eax` (and thus ignore the instruction fetch that must have happened prior).

To **translate** this virtual address that the process generated, we have to first split it into two components: the **virtual page number** (VPN), and the **offset** within the page. For this example, because the virtual address space of the process is 64 bytes, we need 6 bits total for our virtual address ( $2^6 = 64$ ). Thus, our virtual address can be conceptualized as follows:



In this diagram, `Va5` is the highest-order bit of the virtual address, and `Va0` the lowest-order bit. Because we know the page size (16 bytes), we can further divide the virtual address as follows:

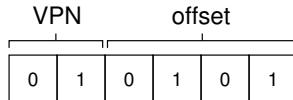


The page size is 16 bytes in a 64-byte address space; thus we need to be able to select 4 pages, and the top 2 bits of the address do just that. Thus, we have a 2-bit virtual page number (VPN). The remaining bits tell us which byte of the page we are interested in, 4 bits in this case; we call this the offset.

When a process generates a virtual address, the OS and hardware must combine to translate it into a meaningful physical address. For example, let us assume the load above was to virtual address 21:

```
movl 21, %eax
```

Turning “21” into binary form, we get “010101”, and thus we can examine this virtual address and see how it breaks down into a virtual page number (VPN) and offset:



Thus, the virtual address “21” is on the 5th (“0101”th) byte of virtual page “01” (or 1). With our virtual page number, we can now index our page table and find which physical frame virtual page 1 resides within. In the page table above the **physical frame number (PFN)** (also sometimes called the **physical page number or PPN**) is 7 (binary 111). Thus, we can translate this virtual address by replacing the VPN with the PFN and then issue the load to physical memory (Figure 18.3).

Note the offset stays the same (i.e., it is not translated), because the offset just tells us which byte *within* the page we want. Our final physical address is 1110101 (117 in decimal), and is exactly where we want our load to fetch data from (Figure 18.2, page 2).

With this basic overview in mind, we can now ask (and hopefully, answer) a few basic questions you may have about paging. For example, where are these page tables stored? What are the typical contents of the page table, and how big are the tables? Does paging make the system (too) slow? These and other beguiling questions are answered, at least in part, in the text below. Read on!

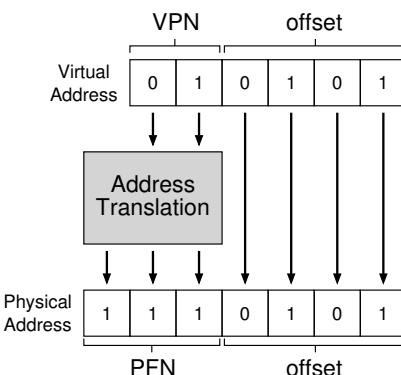


Figure 18.3: The Address Translation Process

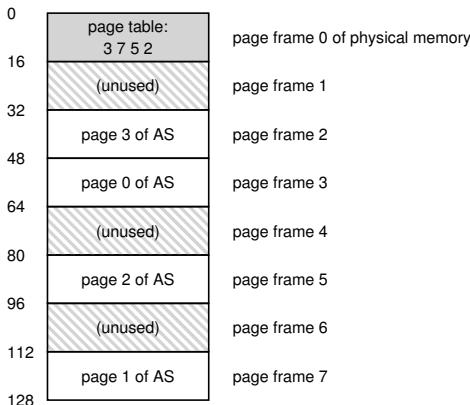


Figure 18.4: Example: Page Table in Kernel Physical Memory

## 18.2 Where Are Page Tables Stored?

Page tables can get terribly large, much bigger than the small segment table or base/bounds pair we have discussed previously. For example, imagine a typical 32-bit address space, with 4KB pages. This virtual address splits into a 20-bit VPN and 12-bit offset (recall that 10 bits would be needed for a 1KB page size, and just add two more to get to 4KB).

A 20-bit VPN implies that there are  $2^{20}$  translations that the OS would have to manage for each process (that's roughly a million); assuming we need 4 bytes per **page table entry (PTE)** to hold the physical translation plus any other useful stuff, we get an immense 4MB of memory needed for each page table! That is pretty large. Now imagine there are 100 processes running: this means the OS would need 400MB of memory just for all those address translations! Even in the modern era, where machines have gigabytes of memory, it seems a little crazy to use a large chunk of it just for translations, no? And we won't even think about how big such a page table would be for a 64-bit address space; that would be too gruesome and perhaps scare you off entirely.

Because page tables are so big, we don't keep any special on-chip hardware in the MMU to store the page table of the currently-running process. Instead, we store the page table for each process in *memory* somewhere. Let's assume for now that the page tables live in physical memory that the OS manages; later we'll see that much of OS memory itself can be virtualized, and thus page tables can be stored in OS virtual memory (and even swapped to disk), but that is too confusing right now, so we'll ignore it. In Figure 18.4 is a picture of a page table in OS memory; see the tiny set of translations in there?

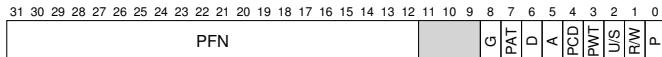


Figure 18.5: An x86 Page Table Entry (PTE)

### 18.3 What's Actually In The Page Table?

Let's talk a little about page table organization. The page table is just a data structure that is used to map virtual addresses (or really, virtual page numbers) to physical addresses (physical frame numbers). Thus, any data structure could work. The simplest form is called a **linear page table**, which is just an array. The OS *indexes* the array by the virtual page number (VPN), and looks up the page-table entry (PTE) at that index in order to find the desired physical frame number (PFN). For now, we will assume this simple linear structure; in later chapters, we will make use of more advanced data structures to help solve some problems with paging.

As for the contents of each PTE, we have a number of different bits in there worth understanding at some level. A **valid bit** is common to indicate whether the particular translation is valid; for example, when a program starts running, it will have code and heap at one end of its address space, and the stack at the other. All the unused space in-between will be marked **invalid**, and if the process tries to access such memory, it will generate a trap to the OS which will likely terminate the process. Thus, the valid bit is crucial for supporting a sparse address space; by simply marking all the unused pages in the address space invalid, we remove the need to allocate physical frames for those pages and thus save a great deal of memory.

We also might have **protection bits**, indicating whether the page could be read from, written to, or executed from. Again, accessing a page in a way not allowed by these bits will generate a trap to the OS.

There are a couple of other bits that are important but we won't talk about much for now. A **present bit** indicates whether this page is in physical memory or on disk (i.e., it has been **swapped out**). We will understand this machinery further when we study how to **swap** parts of the address space to disk to support address spaces that are larger than physical memory; swapping allows the OS to free up physical memory by moving rarely-used pages to disk. A **dirty bit** is also common, indicating whether the page has been modified since it was brought into memory.

A **reference bit** (a.k.a. **accessed bit**) is sometimes used to track whether a page has been accessed, and is useful in determining which pages are popular and thus should be kept in memory; such knowledge is critical during **page replacement**, a topic we will study in great detail in subsequent chapters.

Figure 18.5 shows an example page table entry from the x86 architecture [109]. It contains a present bit (P); a read/write bit (R/W) which determines if writes are allowed to this page; a user/supervisor bit (U/S)

which determines if user-mode processes can access the page; a few bits (PWT, PCD, PAT, and G) that determine how hardware caching works for these pages; an accessed bit (A) and a dirty bit (D); and finally, the page frame number (PFN) itself.

Read the Intel Architecture Manuals [I09] for more details on x86 paging support. Be forewarned, however; reading manuals such as these, while quite informative (and certainly necessary for those who write code to use such page tables in the OS), can be challenging at first. A little patience, and a lot of desire, is required.

## 18.4 Paging: Also Too Slow

With page tables in memory, we already know that they might be too big. As it turns out, they can slow things down too. For example, take our simple instruction:

```
movl 21, %eax
```

Again, let's just examine the explicit reference to address 21 and not worry about the instruction fetch. In this example, we'll assume the hardware performs the translation for us. To fetch the desired data, the system must first **translate** the virtual address (21) into the correct physical address (117). Thus, before fetching the data from address 117, the system must first fetch the proper page table entry from the process's page table, perform the translation, and then load the data from physical memory.

To do so, the hardware must know where the page table is for the currently-running process. Let's assume for now that a single **page-table base register** contains the physical address of the starting location of the page table. To find the location of the desired PTE, the hardware will thus perform the following functions:

```
VPN      = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))
```

In our example, VPN\\_MASK would be set to 0x30 (hex 30, or binary 110000) which picks out the VPN bits from the full virtual address; SHIFT is set to 4 (the number of bits in the offset), such that we move the VPN bits down to form the correct integer virtual page number. For example, with virtual address 21 (010101), and masking turns this value into 010000; the shift turns it into 01, or virtual page 1, as desired. We then use this value as an index into the array of PTEs pointed to by the page table base register.

Once this physical address is known, the hardware can fetch the PTE from memory, extract the PFN, and concatenate it with the offset from the virtual address to form the desired physical address. Specifically, you can think of the PFN being left-shifted by SHIFT, and then logically OR'd with the offset to form the final address as follows:

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

```

1 // Extract the VPN from the virtual address
2 VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3
4 // Form the address of the page-table entry (PTE)
5 PTEAddr = PTBR + (VPN * sizeof(PTE))
6
7 // Fetch the PTE
8 PTE = AccessMemory(PTEAddr)
9
10 // Check if process can access the page
11 if (PTE.Valid == False)
12     RaiseException(SEGMENTATION_FAULT)
13 else if (CanAccess(PTE.ProtectBits) == False)
14     RaiseException(PROTECTION_FAULT)
15 else
16     // Access is OK: form physical address and fetch it
17     offset = VirtualAddress & OFFSET_MASK
18     PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19     Register = AccessMemory(PhysAddr)

```

Figure 18.6: Accessing Memory With Paging

Finally, the hardware can fetch the desired data from memory and put it into register eax. The program has now succeeded at loading a value from memory!

To summarize, we now describe the initial protocol for what happens on each memory reference. Figure 18.6 shows the basic approach. For every memory reference (whether an instruction fetch or an explicit load or store), paging requires us to perform one extra memory reference in order to first fetch the translation from the page table. That is a lot of work! Extra memory references are costly, and in this case will likely slow down the process by a factor of two or more.

And now you can hopefully see that there are *two* real problems that we must solve. Without careful design of both hardware and software, page tables will cause the system to run too slowly, as well as take up too much memory. While seemingly a great solution for our memory virtualization needs, these two crucial problems must first be overcome.

## 18.5 A Memory Trace

Before closing, we now trace through a simple memory access example to demonstrate all of the resulting memory accesses that occur when using paging. The code snippet (in C, in a file called `array.c`) that we are interested in is as follows:

```

int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;

```

We compile `array.c` and run it with the following commands:

**ASIDE: DATA STRUCTURE — THE PAGE TABLE**

One of the most important data structures in the memory management subsystem of a modern OS is the **page table**. In general, a page table stores **virtual-to-physical address translations**, thus letting the system know where each page of an address space actually resides in physical memory. Because each address space requires such translations, in general there is one page table per process in the system. The exact structure of the page table is either determined by the hardware (older systems) or can be more flexibly managed by the OS (modern systems).

```
prompt> gcc -o array array.c -Wall -O
prompt> ./array
```

Of course, to truly understand what memory accesses this code snippet (which simply initializes an array) will make, we'll have to know (or assume) a few more things. First, we'll have to **disassemble** the resulting binary (using `objdump` on Linux, or `otool` on a Mac) to see what assembly instructions are used to initialize the array in a loop. Here is the resulting assembly code:

```
1024 movl $0x0, (%edi,%eax,4)
1028 incl %eax
1032 cmpl $0x03e8,%eax
1036 jne 0x1024
```

The code, if you know a little x86, is actually quite easy to understand<sup>2</sup>. 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.

---

<sup>2</sup>We are cheating a little bit here, assuming each instruction is four bytes in size for simplicity; in actuality, x86 instructions are variable-sized.

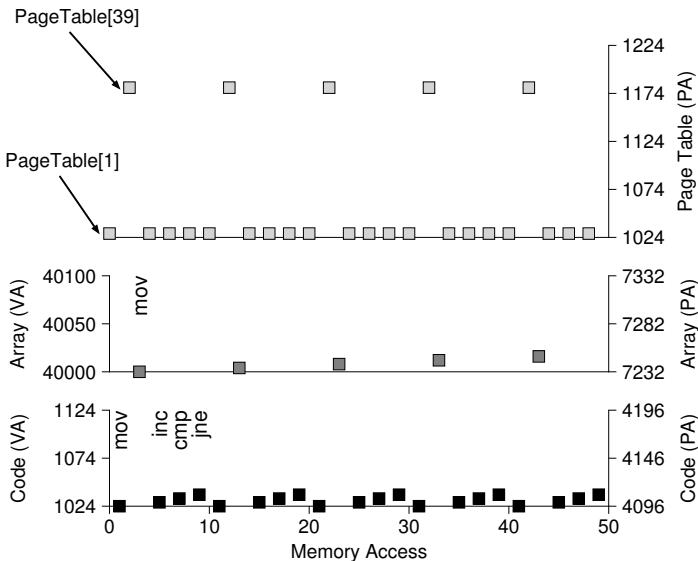


Figure 18.7: A Virtual (And Physical) Memory Trace

All we need to know now are the contents of the page table, and its location in physical memory. Let's assume we have a linear (array-based) page table and that it is located at physical address 1KB (1024).

As for its contents, there are just a few virtual pages we need to worry about having mapped for this example. First, there is the virtual page the code lives on. Because the page size is 1KB, virtual address 1024 resides on the second page of the virtual address space (VPN=1, as VPN=0 is the first page). Let's assume this virtual page maps to physical frame 4 (VPN 1 → 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 → PFN 7), (VPN 40 → PFN 8), (VPN 41 → PFN 9), (VPN 42 → PFN 10).

We are now ready to trace the memory references of the program. When it runs, each instruction fetch will generate two memory references: one to the page table to find the physical frame that the instruction resides within, and one to the instruction itself to fetch it to the CPU for processing. In addition, there is one explicit memory reference in the form of the `mov` instruction; this adds another page table access first (to translate the array virtual address to the correct physical one) and then the array access itself.

The entire process, for the first five loop iterations, is depicted in Figure 18.7 (page 10). The bottom most graph shows the instruction memory references on the y-axis in black (with virtual addresses on the left, and the actual physical addresses on the right); the middle graph shows array accesses in dark gray (again with virtual on left and physical on right); finally, the topmost graph shows page table memory accesses in light gray (just physical, as the page table in this example resides in physical memory). The x-axis, for the entire trace, shows memory accesses across the first five iterations of the loop; there are 10 memory accesses per loop, which includes four instruction fetches, one explicit update of memory, and five page table accesses to translate those four fetches and one explicit update.

See if you can make sense of the patterns that show up in this visualization. In particular, what will change as the loop continues to run beyond these first five iterations? Which new memory locations will be accessed? Can you figure it out?

This has just been the simplest of examples (only a few lines of C code), and yet you might already be able to sense the complexity of understanding the actual memory behavior of real applications. Don't worry: it definitely gets worse, because the mechanisms we are about to introduce only complicate this already complex machinery. Sorry<sup>3</sup>!

## 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.

---

<sup>3</sup>We're not really sorry. But, we are sorry about not being sorry, if that makes sense.

## References

- [KE+62] "One-level Storage System"  
T. Kilburn, and D.B.G. Edwards and M.J. Lanigan and F.H. Sumner  
IRE Trans. EC-11, 2 (1962), pp. 223-235  
(Reprinted in Bell and Newell, "Computer Structures: Readings and Examples" McGraw-Hill, New York, 1971).  
*The Atlas pioneered the idea of dividing memory into fixed-sized pages and in many senses was an early form of the memory-management ideas we see in modern computer systems.*
- [I09] "Intel 64 and IA-32 Architectures Software Developer's Manuals"  
Intel, 2009  
Available: <http://www.intel.com/products/processor/manuals>  
*In particular, pay attention to "Volume 3A: System Programming Guide Part 1" and "Volume 3B: System Programming Guide Part 2"*
- [L78] "The Manchester Mark I and atlas: a historical perspective"  
S. H. Lavington  
Communications of the ACM archive  
Volume 21, Issue 1 (January 1978), pp. 4-12  
Special issue on computer architecture  
*This paper is a great retrospective of some of the history of the development of some important computer systems. As we sometimes forget in the US, many of these new ideas came from overseas.*

## Homework

In this homework, you will use a simple program, which is known as `paging-linear-translate.py`, to see if you understand how simple virtual-to-physical address translation works with linear page tables. See the README for details.

### Questions

1. Before doing any translations, let's use the simulator to study how linear page tables change size given different parameters. Compute the size of linear page tables as different parameters change. Some suggested inputs are below; by using the `-v` flag, you can see how many page-table entries are filled.

First, to understand how linear page table size changes as the address space grows:

```
paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0  
paging-linear-translate.py -P 1k -a 2m -p 512m -v -n 0  
paging-linear-translate.py -P 1k -a 4m -p 512m -v -n 0
```

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

```
paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0  
paging-linear-translate.py -P 2k -a 1m -p 512m -v -n 0  
paging-linear-translate.py -P 4k -a 1m -p 512m -v -n 0
```

Before running any of these, try to think about the expected trends. How should page-table size change as the address space grows? As the page size grows? Why shouldn't we just use really big pages in general?

2. Now let's do some translations. Start with some small examples, and change the number of pages that are allocated to the address space with the `-u` flag. For example:

```
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 0  
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 25  
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 50  
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 75  
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 100
```

What happens as you increase the percentage of pages that are allocated in each address space?

3. Now let's try some different random seeds, and some different (and sometimes quite crazy) address-space parameters, for variety:

```
paging-linear-translate.py -P 8 -a 32 -p 1024 -v -s 1  
paging-linear-translate.py -P 8k -a 32k -p 1m -v -s 2  
paging-linear-translate.py -P 1m -a 256m -p 512m -v -s 3
```

Which of these parameter combinations are unrealistic? Why?

4. Use the program to try out some other problems. Can you find the limits of where the program doesn't work anymore? For example, what happens if the address-space size is *bigger* than physical memory?

## Paging: Faster Translations (TLBs)

Using paging as the core mechanism to support virtual memory can lead to high performance overheads. By chopping the address space into small, fixed-sized units (i.e., pages), paging requires a large amount of mapping information. Because that mapping information is generally stored in physical memory, paging logically requires an extra memory lookup for each virtual address generated by the program. Going to memory for translation information before every instruction fetch or explicit load or store is prohibitively slow. And thus our problem:

THE CRUX:  
HOW TO SPEED UP ADDRESS TRANSLATION

How can we speed up address translation, and generally avoid the extra memory reference that paging seems to require? What hardware support is required? What OS involvement is needed?

When we want to make things fast, the OS usually needs some help. And help often comes from the OS's old friend: the hardware. To speed address translation, we are going to add what is called (for historical reasons [CP78]) a **translation-lookaside buffer**, or **TLB** [C68, C95]. A TLB is part of the chip's **memory-management unit (MMU)**, and is simply a hardware **cache** of popular virtual-to-physical address translations; thus, a better name would be an **address-translation cache**. Upon each virtual memory reference, the hardware first checks the TLB to see if the desired translation is held therein; if so, the translation is performed (quickly) *without* having to consult the page table (which has all translations). Because of their tremendous performance impact, TLBs in a real sense make virtual memory possible [C95].

### 19.1 TLB Basic Algorithm

Figure 19.1 shows a rough sketch of how hardware might handle a virtual address translation, assuming a simple **linear page table** (i.e., the page table is an array) and a **hardware-managed TLB** (i.e., the hardware handles much of the responsibility of page table accesses; we'll explain more about this below).

```

1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)    // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset   = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10     else           // TLB Miss
11         PTEAddr = PTBR + (VPN * sizeof(PTE))
12         PTE = AccessMemory(PTEAddr)
13         if (PTE.Valid == False)
14             RaiseException(SEGMENTATION_FAULT)
15         else if (CanAccess(PTE.ProtectBits) == False)
16             RaiseException(PROTECTION_FAULT)
17         else
18             TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19         RetryInstruction()

```

Figure 19.1: **TLB Control Flow Algorithm**

The algorithm the hardware follows works like this: first, extract the virtual page number (VPN) from the virtual address (Line 1 in Figure 19.1), and check if the TLB holds the translation for this VPN (Line 2). If it does, we have a **TLB hit**, which means the TLB holds the translation. Success! We can now extract the page frame number (PFN) from the relevant TLB entry, concatenate that onto the offset from the original virtual address, and form the desired physical address (PA), and access memory (Lines 5–7), assuming protection checks do not fail (Line 4).

If the CPU does not find the translation in the TLB (a **TLB miss**), we have some more work to do. In this example, the hardware accesses the page table to find the translation (Lines 11–12), and, assuming that the virtual memory reference generated by the process is valid and accessible (Lines 13, 15), updates the TLB with the translation (Line 18). These set of actions are costly, primarily because of the extra memory reference needed to access the page table (Line 12). Finally, once the TLB is updated, the hardware retries the instruction; this time, the translation is found in the TLB, and the memory reference is processed quickly.

The TLB, like all caches, is built on the premise that in the common case, translations are found in the cache (i.e., are hits). If so, little overhead is added, as the TLB is found near the processing core and is designed to be quite fast. When a miss occurs, the high cost of paging is incurred; the page table must be accessed to find the translation, and an extra memory reference (or more, with more complex page tables) results. If this happens often, the program will likely run noticeably more slowly; memory accesses, relative to most CPU instructions, are quite costly, and TLB misses lead to more memory accesses. Thus, it is our hope to avoid TLB misses as much as we can.

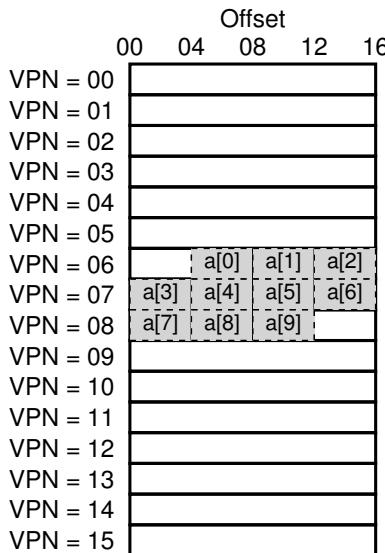


Figure 19.2: Example: An Array In A Tiny Address Space

## 19.2 Example: Accessing An Array

To make clear the operation of a TLB, let's examine a simple virtual address trace and see how a TLB can improve its performance. In this example, let's assume we have an array of 10 4-byte integers in memory, starting at virtual address 100. Assume further that we have a small 8-bit virtual address space, with 16-byte pages; thus, a virtual address breaks down into a 4-bit VPN (there are 16 virtual pages) and a 4-bit offset (there are 16 bytes on each of those pages).

Figure 19.2 shows the array laid out on the 16 16-byte pages of the system. As you can see, the array's first entry ( $a[0]$ ) begins on (VPN=06, offset=04); only three 4-byte integers fit onto that page. The array continues onto the next page (VPN=07), where the next four entries ( $a[3] \dots a[6]$ ) are found. Finally, the last three entries of the 10-entry array ( $a[7] \dots a[9]$ ) are located on the next page of the address space (VPN=08).

Now let's consider a simple loop that accesses each array element, something that would look like this in C:

```
int sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i];
}
```

For the sake of simplicity, we will pretend that the only memory accesses the loop generates are to the array (ignoring the variables *i* and *sum*, as well as the instructions themselves). When the first array element ( $a[0]$ ) is accessed, the CPU will see a load to virtual address 100. The hardware extracts the VPN from this (VPN=06), and uses that to check the TLB for a valid translation. Assuming this is the first time the program accesses the array, the result will be a TLB miss.

The next access is to  $a[1]$ , and there is some good news here: a TLB hit! Because the second element of the array is packed next to the first, it lives on the same page; because we've already accessed this page when accessing the first element of the array, the translation is already loaded into the TLB. And hence the reason for our success. Access to  $a[2]$  encounters similar success (another hit), because it too lives on the same page as  $a[0]$  and  $a[1]$ .

Unfortunately, when the program accesses  $a[3]$ , we encounter another TLB miss. However, once again, the next entries ( $a[4] \dots a[6]$ ) will hit in the TLB, as they all reside on the same page in memory.

Finally, access to  $a[7]$  causes one last TLB miss. The hardware once again consults the page table to figure out the location of this virtual page in physical memory, and updates the TLB accordingly. The final two accesses ( $a[8]$  and  $a[9]$ ) receive the benefits of this TLB update; when the hardware looks in the TLB for their translations, two more hits result.

Let us summarize TLB activity during our ten accesses to the array: **miss**, hit, hit, **miss**, hit, hit, hit, **miss**, hit, hit. Thus, our TLB **hit rate**, which is the number of hits divided by the total number of accesses, is 70%. Although this is not too high (indeed, we desire hit rates that approach 100%), it is non-zero, which may be a surprise. Even though this is the first time the program accesses the array, the TLB improves performance due to **spatial locality**. The elements of the array are packed tightly into pages (i.e., they are close to one another in **space**), and thus only the first access to an element on a page yields a TLB miss.

Also note the role that page size plays in this example. If the page size had simply been twice as big (32 bytes, not 16), the array access would suffer even fewer misses. As typical page sizes are more like 4KB, these types of dense, array-based accesses achieve excellent TLB performance, encountering only a single miss per page of accesses.

One last point about TLB performance: if the program, soon after this loop completes, accesses the array again, we'd likely see an even better result, assuming that we have a big enough TLB to cache the needed translations: hit, hit, hit, hit, hit, hit, hit, hit, hit. In this case, the TLB hit rate would be high because of **temporal locality**, i.e., the quick re-referencing of memory items in **time**. Like any cache, TLBs rely upon both spatial and temporal locality for success, which are program properties. If the program of interest exhibits such locality (and many programs do), the TLB hit rate will likely be high.

**TIP: USE CACHING WHEN POSSIBLE**

Caching is one of the most fundamental performance techniques in computer systems, one that is used again and again to make the “common-case fast” [HP06]. The idea behind hardware caches is to take advantage of **locality** in instruction and data references. There are usually two types of locality: **temporal locality** and **spatial locality**. With temporal locality, the idea is that an instruction or data item that has been recently accessed will likely be re-accessed soon in the future. Think of loop variables or instructions in a loop; they are accessed repeatedly over time. With spatial locality, the idea is that if a program accesses memory at address  $x$ , it will likely soon access memory near  $x$ . Imagine here streaming through an array of some kind, accessing one element and then the next. Of course, these properties depend on the exact nature of the program, and thus are not hard-and-fast laws but more like rules of thumb.

Hardware caches, whether for instructions, data, or address translations (as in our TLB) take advantage of locality by keeping copies of memory in small, fast on-chip memory. Instead of having to go to a (slow) memory to satisfy a request, the processor can first check if a nearby copy exists in a cache; if it does, the processor can access it quickly (i.e., in a few CPU cycles) and avoid spending the costly time it takes to access memory (many nanoseconds).

You might be wondering: if caches (like the TLB) are so great, why don’t we just make bigger caches and keep all of our data in them? Unfortunately, this is where we run into more fundamental laws like those of physics. If you want a fast cache, it has to be small, as issues like the speed-of-light and other physical constraints become relevant. Any large cache by definition is slow, and thus defeats the purpose. Thus, we are stuck with small, fast caches; the question that remains is how to best use them to improve performance.

### 19.3 Who Handles The TLB Miss?

One question that we must answer: who handles a TLB miss? Two answers are possible: the hardware, or the software (OS). In the olden days, the hardware had complex instruction sets (sometimes called **CISC**, for complex-instruction set computers) and the people who built the hardware didn’t much trust those sneaky OS people. Thus, the hardware would handle the TLB miss entirely. To do this, the hardware has to know exactly *where* the page tables are located in memory (via a **page-table base register**, used in Line 11 in Figure 19.1), as well as their *exact format*; on a miss, the hardware would “walk” the page table, find the correct page-table entry and extract the desired translation, update the TLB with the translation, and retry the instruction. An example of an “older” architecture that has **hardware-managed TLBs** is the Intel x86 architecture, which uses a fixed **multi-level page table** (see the next chapter for details); the current page table is pointed to by the CR3 register [I09].

```

1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)    // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10     else           // TLB Miss
11         RaiseException(TLB_MISS)

```

Figure 19.3: TLB Control Flow Algorithm (OS Handled)

More modern architectures (e.g., MIPS R10k [H93] or Sun’s SPARC v9 [WG00], both **RISC** or reduced-instruction set computers) have what is known as a **software-managed TLB**. On a TLB miss, the hardware simply raises an exception (line 11 in Figure 19.3), which pauses the current instruction stream, raises the privilege level to kernel mode, and jumps to a **trap handler**. As you might guess, this trap handler is code within the OS that is written with the express purpose of handling TLB misses. When run, the code will lookup the translation in the page table, use special “privileged” instructions to update the TLB, and return from the trap; at this point, the hardware retries the instruction (resulting in a TLB hit).

Let’s discuss a couple of important details. First, the return-from-trap instruction needs to be a little different than the return-from-trap we saw before when servicing a system call. In the latter case, the return-from-trap should resume execution at the instruction *after* the trap into the OS, just as a return from a procedure call returns to the instruction immediately following the call into the procedure. In the former case, when returning from a TLB miss-handling trap, the hardware must resume execution at the instruction that *caused* the trap; this retry thus lets the instruction run again, this time resulting in a TLB hit. Thus, depending on how a trap or exception was caused, the hardware must save a different PC when trapping into the OS, in order to resume properly when the time to do so arrives.

Second, when running the TLB miss-handling code, the OS needs to be extra careful not to cause an infinite chain of TLB misses to occur. Many solutions exist; for example, you could keep TLB miss handlers in physical memory (where they are **unmapped** and not subject to address translation), or reserve some entries in the TLB for permanently-valid translations and use some of those permanent translation slots for the handler code itself; these **wired** translations always hit in the TLB.

The primary advantage of the software-managed approach is *flexibility*: the OS can use any data structure it wants to implement the page table, without necessitating hardware change. Another advantage is *simplicity*; as you can see in the TLB control flow (line 11 in Figure 19.3, in contrast to lines 11–19 in Figure 19.1), the hardware doesn’t have to do much on a miss; it raises an exception, and the OS TLB miss handler does the rest.

### ASIDE: RISC vs. CISC

In the 1980's, a great battle took place in the computer architecture community. On one side was the **CISC** camp, which stood for **Complex Instruction Set Computing**; on the other side was **RISC**, for **Reduced Instruction Set Computing** [PS81]. The RISC side was spear-headed by David Patterson at Berkeley and John Hennessy at Stanford (who are also co-authors of some famous books [HP06]), although later John Cocke was recognized with a Turing award for his earliest work on RISC [CM00]. CISC instruction sets tend to have a lot of instructions in them, and each instruction is relatively powerful. For example, you might see a string copy, which takes two pointers and a length and copies bytes from source to destination. The idea behind CISC was that instructions should be high-level primitives, to make the assembly language itself easier to use, and to make code more compact.

RISC instruction sets are exactly the opposite. A key observation behind RISC is that instruction sets are really compiler targets, and all compilers really want are a few simple primitives that they can use to generate high-performance code. Thus, RISC proponents argued, let's rip out as much from the hardware as possible (especially the microcode), and make what's left simple, uniform, and fast.

In the early days, RISC chips made a huge impact, as they were noticeably faster [BC91]; many papers were written; a few companies were formed (e.g., MIPS and Sun). However, as time progressed, CISC manufacturers such as Intel incorporated many RISC techniques into the core of their processors, for example by adding early pipeline stages that transformed complex instructions into micro-instructions which could then be processed in a RISC-like manner. These innovations, plus a growing number of transistors on each chip, allowed CISC to remain competitive. The end result is that the debate died down, and today both types of processors can be made to run fast.

## 19.4 TLB Contents: What's In There?

Let's look at the contents of the hardware TLB in more detail. A typical TLB might have 32, 64, or 128 entries and be what is called **fully associative**. Basically, this just means that any given translation can be anywhere in the TLB, and that the hardware will search the entire TLB in parallel to find the desired translation. A TLB entry might look like this:

VPN | PFN | other bits

Note that both the VPN and PFN are present in each entry, as a translation could end up in any of these locations (in hardware terms, the TLB is known as a **fully-associative** cache). The hardware searches the entries in parallel to see if there is a match.

**ASIDE: TLB VALID BIT  $\neq$  PAGE TABLE VALID BIT**

A common mistake is to confuse the valid bits found in a TLB with those found in a page table. In a page table, when a page-table entry (PTE) is marked invalid, it means that the page has not been allocated by the process, and should not be accessed by a correctly-working program. The usual response when an invalid page is accessed is to trap to the OS, which will respond by killing the process.

A TLB valid bit, in contrast, simply refers to whether a TLB entry has a valid translation within it. When a system boots, for example, a common initial state for each TLB entry is to be set to invalid, because no address translations are yet cached there. Once virtual memory is enabled, and once programs start running and accessing their virtual address spaces, the TLB is slowly populated, and thus valid entries soon fill the TLB.

The TLB valid bit is quite useful when performing a context switch too, as we'll discuss further below. By setting all TLB entries to invalid, the system can ensure that the about-to-be-run process does not accidentally use a virtual-to-physical translation from a previous process.

More interesting are the “other bits”. For example, the TLB commonly has a **valid** bit, which says whether the entry has a valid translation or not. Also common are **protection** bits, which determine how a page can be accessed (as in the page table). For example, code pages might be marked *read and execute*, whereas heap pages might be marked *read and write*. There may also be a few other fields, including an **address-space identifier**, a **dirty bit**, and so forth; see below for more information.

## 19.5 TLB Issue: Context Switches

With TLBs, some new issues arise when switching between processes (and hence address spaces). Specifically, the TLB contains virtual-to-physical translations that are only valid for the currently running process; these translations are not meaningful for other processes. As a result, when switching from one process to another, the hardware or OS (or both) must be careful to ensure that the about-to-be-run process does not accidentally use translations from some previously run process.

To understand this situation better, let's look at an example. When one process (P1) is running, it assumes the TLB might be caching translations that are valid for it, i.e., that come from P1's page table. Assume, for this example, that the 10th virtual page of P1 is mapped to physical frame 100.

In this example, assume another process (P2) exists, and the OS soon might decide to perform a context switch and run it. Assume here that the 10th virtual page of P2 is mapped to physical frame 170. If entries for both processes were in the TLB, the contents of the TLB would be:

VPN	PFN	valid	prot
10	100	1	rwx
—	—	0	—
10	170	1	rwx
—	—	0	—

In the TLB above, we clearly have a problem: VPN 10 translates to either PFN 100 (P1) or PFN 170 (P2), but the hardware can't distinguish which entry is meant for which process. Thus, we need to do some more work in order for the TLB to correctly and efficiently support virtualization across multiple processes. And thus, a crux:

#### THE CRUX:

#### HOW TO MANAGE TLB CONTENTS ON A CONTEXT SWITCH

When context-switching between processes, the translations in the TLB for the last process are not meaningful to the about-to-be-run process. What should the hardware or OS do in order to solve this problem?

There are a number of possible solutions to this problem. One approach is to simply **flush** the TLB on context switches, thus emptying it before running the next process. On a software-based system, this can be accomplished with an explicit (and privileged) hardware instruction; with a hardware-managed TLB, the flush could be enacted when the page-table base register is changed (note the OS must change the PTBR on a context switch anyhow). In either case, the flush operation simply sets all valid bits to 0, essentially clearing the contents of the TLB.

By flushing the TLB on each context switch, we now have a working solution, as a process will never accidentally encounter the wrong translations in the TLB. However, there is a cost: each time a process runs, it must incur TLB misses as it touches its data and code pages. If the OS switches between processes frequently, this cost may be high.

To reduce this overhead, some systems add hardware support to enable sharing of the TLB across context switches. In particular, some hardware systems provide an **address space identifier (ASID)** field in the TLB. You can think of the ASID as a **process identifier (PID)**, but usually it has fewer bits (e.g., 8 bits for the ASID versus 32 bits for a PID).

If we take our example TLB from above and add ASIDs, it is clear processes can readily share the TLB: only the ASID field is needed to differentiate otherwise identical translations. Here is a depiction of a TLB with the added ASID field:

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

Thus, with address-space identifiers, the TLB can hold translations from different processes at the same time without any confusion. Of course, the hardware also needs to know which process is currently running in order to perform translations, and thus the OS must, on a context switch, set some privileged register to the ASID of the current process.

As an aside, you may also have thought of another case where two entries of the TLB are remarkably similar. In this example, there are two entries for two different processes with two different VPNs that point to the *same* physical page:

VPN	PFN	valid	prot	ASID
10	101	1	r-x	1
—	—	0	—	—
50	101	1	r-x	2
—	—	0	—	—

This situation might arise, for example, when two processes *share* a page (a code page, for example). In the example above, Process 1 is sharing physical page 101 with Process 2; P1 maps this page into the 10th page of its address space, whereas P2 maps it to the 50th page of its address space. Sharing of code pages (in binaries, or shared libraries) is useful as it reduces the number of physical pages in use, thus reducing memory overheads.

## 19.6 Issue: Replacement Policy

As with any cache, and thus also with the TLB, one more issue that we must consider is **cache replacement**. Specifically, when we are installing a new entry in the TLB, we have to **replace** an old one, and thus the question: which one to replace?

### THE CRUX: HOW TO DESIGN TLB REPLACEMENT POLICY

Which TLB entry should be replaced when we add a new TLB entry? The goal, of course, being to minimize the **miss rate** (or increase **hit rate**) and thus improve performance.

We will study such policies in some detail when we tackle the problem of swapping pages to disk; here we'll just highlight a few typical policies. One common approach is to evict the **least-recently-used** or **LRU** entry. LRU tries to take advantage of locality in the memory-reference stream, assuming it is likely that an entry that has not recently been used is a good candidate for eviction. Another typical approach is to use a **random** policy, which evicts a TLB mapping at random. Such a policy is useful due to its simplicity and ability to avoid corner-case behaviors; for example, a “reasonable” policy such as LRU behaves quite unreasonably when a program loops over  $n + 1$  pages with a TLB of size  $n$ ; in this case, LRU misses upon every access, whereas random does much better.



Figure 19.4: A MIPS TLB Entry

## 19.7 A Real TLB Entry

Finally, let's briefly look at a real TLB. This example is from the MIPS R4000 [H93], a modern system that uses software-managed TLBs; a slightly simplified MIPS TLB entry can be seen in Figure 19.4.

The MIPS R4000 supports a 32-bit address space with 4KB pages. Thus, we would expect a 20-bit VPN and 12-bit offset in our typical virtual address. However, as you can see in the TLB, there are only 19 bits for the VPN; as it turns out, user addresses will only come from half the address space (the rest reserved for the kernel) and hence only 19 bits of VPN are needed. The VPN translates to up to a 24-bit physical frame number (PFN), and hence can support systems with up to 64GB of (physical) main memory ( $2^{24}$  4KB pages).

There are a few other interesting bits in the MIPS TLB. We see a *global* bit (G), which is used for pages that are globally-shared among processes. Thus, if the global bit is set, the ASID is ignored. We also see the 8-bit *ASID*, which the OS can use to distinguish between address spaces (as described above). One question for you: what should the OS do if there are more than 256 ( $2^8$ ) processes running at a time? Finally, we see 3 *Coherence* (C) bits, which determine how a page is cached by the hardware (a bit beyond the scope of these notes); a *dirty* bit which is marked when the page has been written to (we'll see the use of this later); a *valid* bit which tells the hardware if there is a valid translation present in the entry. There is also a *page mask* field (not shown), which supports multiple page sizes; we'll see later why having larger pages might be useful. Finally, some of the 64 bits are unused (shaded gray in the diagram).

MIPS TLBs usually have 32 or 64 of these entries, most of which are used by user processes as they run. However, a few are reserved for the OS. A *wired* register can be set by the OS to tell the hardware how many slots of the TLB to reserve for the OS; the OS uses these reserved mappings for code and data that it wants to access during critical times, where a TLB miss would be problematic (e.g., in the TLB miss handler).

Because the MIPS TLB is software managed, there needs to be instructions to update the TLB. The MIPS provides four such instructions: *TLBP*, which probes the TLB to see if a particular translation is in there; *TLBR*, which reads the contents of a TLB entry into registers; *TLBWI*, which replaces a specific TLB entry; and *TLBWR*, which replaces a random TLB entry. The OS uses these instructions to manage the TLB's contents. It is of course critical that these instructions are **privileged**; imagine what a user process could do if it could modify the contents of the TLB (hint: just about anything, including take over the machine, run its own malicious "OS", or even make the Sun disappear).

**TIP: RAM ISN'T ALWAYS RAM (CULLER'S LAW)**

The term **random-access memory**, or **RAM**, implies that you can access any part of RAM just as quickly as another. While it is generally good to think of RAM in this way, because of hardware/OS features such as the TLB, accessing a particular page of memory may be costly, particularly if that page isn't currently mapped by your TLB. Thus, it is always good to remember the implementation tip: **RAM isn't always RAM**. Sometimes randomly accessing your address space, particular if the number of pages accessed exceeds the TLB coverage, can lead to severe performance penalties. Because one of our advisors, David Culler, used to always point to the TLB as the source of many performance problems, we name this law in his honor: **Culler's Law**.

## 19.8 Summary

We have seen how hardware can help us make address translation faster. By providing a small, dedicated on-chip TLB as an address-translation cache, most memory references will hopefully be handled *without* having to access the page table in main memory. Thus, in the common case, the performance of the program will be almost as if memory isn't being virtualized at all, an excellent achievement for an operating system, and certainly essential to the use of paging in modern systems.

However, TLBs do not make the world rosy for every program that exists. In particular, if the number of pages a program accesses in a short period of time exceeds the number of pages that fit into the TLB, the program will generate a large number of TLB misses, and thus run quite a bit more slowly. We refer to this phenomenon as exceeding the **TLB coverage**, and it can be quite a problem for certain programs. One solution, as we'll discuss in the next chapter, is to include support for larger page sizes; by mapping key data structures into regions of the program's address space that are mapped by larger pages, the effective coverage of the TLB can be increased. Support for large pages is often exploited by programs such as a **database management system** (a **DBMS**), which have certain data structures that are both large and randomly-accessed.

One other TLB issue worth mentioning: TLB access can easily become a bottleneck in the CPU pipeline, in particular with what is called a **physically-indexed cache**. With such a cache, address translation has to take place *before* the cache is accessed, which can slow things down quite a bit. Because of this potential problem, people have looked into all sorts of clever ways to access caches with *virtual* addresses, thus avoiding the expensive step of translation in the case of a cache hit. Such a **virtually-indexed cache** solves some performance problems, but introduces new issues into hardware design as well. See Wiggins's fine survey for more details [W03].

## References

- [BC91] "Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization"  
 D. Bhandarkar and Douglas W. Clark  
*Communications of the ACM, September 1991*  
*A great and fair comparison between RISC and CISC. The bottom line: on similar hardware, RISC was about a factor of three better in performance.*
- [CM00] "The evolution of RISC technology at IBM"  
 John Cocke and V. Markstein  
*IBM Journal of Research and Development, 44:1/2*  
*A summary of the ideas and work behind the IBM 801, which many consider the first true RISC microprocessor.*
- [C95] "The Core of the Black Canyon Computer Corporation"  
 John Couleur  
*IEEE Annals of History of Computing, 17:4, 1995*  
*In this fascinating historical note, Couleur talks about how he invented the TLB in 1964 while working for GE, and the fortuitous collaboration that thus ensued with the Project MAC folks at MIT.*
- [CG68] "Shared-access Data Processing System"  
 John F. Couleur and Edward L. Glaser  
*Patent 3412382, November 1968*  
*The patent that contains the idea for an associative memory to store address translations. The idea, according to Couleur, came in 1964.*
- [CP78] "The architecture of the IBM System/370"  
 R.P. Case and A. Padegs  
*Communications of the ACM. 21:1, 73-96, January 1978*  
*Perhaps the first paper to use the term **translation lookaside buffer**. The name arises from the historical name for a cache, which was a **lookaside buffer** as called by those developing the Atlas system at the University of Manchester; a cache of address translations thus became a **translation lookaside buffer**. Even though the term lookaside buffer fell out of favor, TLB seems to have stuck, for whatever reason.*
- [H93] "MIPS R4000 Microprocessor User's Manual".  
 Joe Heinrich, Prentice-Hall, June 1993  
 Available: <http://cag.csail.mit.edu/raw/documents/R4400.Uman.book.Ed2.pdf>
- [HP06] "Computer Architecture: A Quantitative Approach"  
 John Hennessy and David Patterson  
 Morgan-Kaufmann, 2006  
*A great book about computer architecture. We have a particular attachment to the classic first edition.*
- [I09] "Intel 64 and IA-32 Architectures Software Developer's Manuals"  
 Intel, 2009  
 Available: <http://www.intel.com/products/processor/manuals>  
*In particular, pay attention to "Volume 3A: System Programming Guide Part 1" and "Volume 3B: System Programming Guide Part 2"*
- [PS81] "RISC-I: A Reduced Instruction Set VLSI Computer"  
 D.A. Patterson and C.H. Sequin  
*ISCA '81, Minneapolis, May 1981*  
*The paper that introduced the term RISC, and started the avalanche of research into simplifying computer chips for performance.*

[SB92] "CPU Performance Evaluation and Execution Time Prediction  
Using Narrow Spectrum Benchmarking"

Rafael H. Saavedra-Barrera

EECS Department, University of California, Berkeley

Technical Report No. UCB/CSD-92-684, February 1992

[www.eecs.berkeley.edu/Pubs/TechRpts/1992/CSD-92-684.pdf](http://www.eecs.berkeley.edu/Pubs/TechRpts/1992/CSD-92-684.pdf)

*A great dissertation about how to predict execution time of applications by breaking them down into constituent pieces and knowing the cost of each piece. Probably the most interesting part that comes out of this work is the tool to measure details of the cache hierarchy (described in Chapter 5). Make sure to check out the wonderful diagrams therein.*

[W03] "A Survey on the Interaction Between Caching, Translation and Protection"

Adam Wiggins

University of New South Wales TR UNSW-CSE-TR-0321, August, 2003

*An excellent survey of how TLBs interact with other parts of the CPU pipeline, namely hardware caches.*

[WG00] "The SPARC Architecture Manual: Version 9"

David L. Weaver and Tom Germond, September 2000

SPARC International, San Jose, California

Available: <http://www.sparc.org/standards/SPARCV9.pdf>

## Homework (Measurement)

In this homework, you are to measure the size and cost of accessing a TLB. The idea is based on work by Saavedra-Barrera [SB92], who developed a simple but beautiful method to measure numerous aspects of cache hierarchies, all with a very simple user-level program. Read his work for more details.

The basic idea is to access some number of pages within a large data structure (e.g., an array) and to time those accesses. For example, let's say the TLB size of a machine happens to be 4 (which would be very small, but useful for the purposes of this discussion). If you write a program that touches 4 or fewer pages, each access should be a TLB hit, and thus relatively fast. However, once you touch 5 pages or more, repeatedly in a loop, each access will suddenly jump in cost, to that of a TLB miss.

The basic code to loop through an array once should look like this:

```
int jump = PAGESIZE / sizeof(int);
for (i = 0; i < NUMPAGES * jump; i += jump) {
    a[i] += 1;
}
```

In this loop, one integer per page of the array `a` is updated, up to the number of pages specified by `NUMPAGES`. By timing such a loop repeatedly (say, a few hundred million times in another loop around this one, or however many loops are needed to run for a few seconds), you can time how long each access takes (on average). By looking for jumps in cost as `NUMPAGES` increases, you can roughly determine how big the first-level TLB is, determine whether a second-level TLB exists (and how big it is if it does), and in general get a good sense of how TLB hits and misses can affect performance.

Figure 19.5 (page 16) shows the average time per access as the number of pages accessed in the loop is increased. As you can see in the graph, when just a few pages are accessed (8 or fewer), the average access time is roughly 5 nanoseconds. When 16 or more pages are accessed, there is a sudden jump to about 20 nanoseconds per access. A final jump in cost occurs at around 1024 pages, at which point each access takes around 70 nanoseconds. From this data, we can conclude that there is a two-level TLB hierarchy; the first is quite small (probably holding between 8 and 16 entries); the second is larger but slower (holding roughly 512 entries). The overall difference between hits in the first-level TLB and misses is quite large, roughly a factor of fourteen. TLB performance matters!

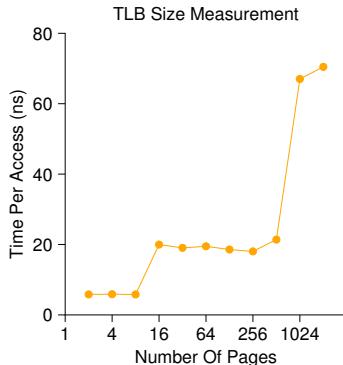


Figure 19.5: Discovering TLB Sizes and Miss Costs

## Questions

1. For timing, you'll need to use a timer such as that made available by `gettimeofday()`. How precise is such a timer? How long does an operation have to take in order for you to time it precisely? (this will help determine how many times, in a loop, you'll have to repeat a page access in order to time it successfully)
2. Write the program, called `tlb.c`, that can roughly measure the cost of accessing each page. Inputs to the program should be: the number of pages to touch and the number of trials.
3. Now write a script in your favorite scripting language (`csh`, `python`, etc.) to run this program, while varying the number of pages accessed from 1 up to a few thousand, perhaps incrementing by a factor of two per iteration. Run the script on different machines and gather some data. How many trials are needed to get reliable measurements?
4. Next, graph the results, making a graph that looks similar to the one above. Use a good tool like `ploticus`. Visualization usually makes the data much easier to digest; why do you think that is?
5. One thing to watch out for is compiler 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?

## Beyond Physical Memory: Mechanisms

Thus far, we've assumed that an address space is unrealistically small and fits into physical memory. In fact, we've been assuming that *every* address space of every running process fits into memory. We will now relax these big assumptions, and assume that we wish to support many concurrently-running large address spaces.

To do so, we require an additional level in the **memory hierarchy**. Thus far, we have assumed that all pages reside in physical memory. However, to support large address spaces, the OS will need a place to stash away portions of address spaces that currently aren't in great demand. In general, the characteristics of such a location are that it should have more capacity than memory; as a result, it is generally slower (if it were faster, we would just use it as memory, no?). In modern systems, this role is usually served by a **hard disk drive**. Thus, in our memory hierarchy, big and slow hard drives sit at the bottom, with memory just above. And thus we arrive at the crux of the problem:

### THE CRUX: HOW TO GO BEYOND PHYSICAL MEMORY

How can the OS make use of a larger, slower device to transparently provide the illusion of a large virtual address space?

One question you might have: why do we want to support a single large address space for a process? Once again, the answer is convenience and ease of use. With a large address space, you don't have to worry about if there is room enough in memory for your program's data structures; rather, you just write the program naturally, allocating memory as needed. It is a powerful illusion that the OS provides, and makes your life vastly simpler. You're welcome! A contrast is found in older systems that used **memory overlays**, which required programmers to manually move pieces of code or data in and out of memory as they were needed [D97]. Try imagining what this would be like: before calling a function or accessing some data, you need to first arrange for the code or data to be in memory; yuck!

**ASIDE: STORAGE TECHNOLOGIES**

We'll delve much more deeply into how I/O devices actually work later (see the chapter on I/O devices). So be patient! And of course the slower device need not be a hard disk, but could be something more modern such as a Flash-based SSD. We'll talk about those things too. For now, just assume we have a big and relatively-slow device which we can use to help us build the illusion of a very large virtual memory, even bigger than physical memory itself.

Beyond just a single process, the addition of swap space allows the OS to support the illusion of a large virtual memory for multiple concurrently-running processes. The invention of multiprogramming (running multiple programs "at once", to better utilize the machine) almost demanded the ability to swap out some pages, as early machines clearly could not hold all the pages needed by all processes at once. Thus, the combination of multiprogramming and ease-of-use leads us to want to support using more memory than is physically available. It is something that all modern VM systems do; it is now something we will learn more about.

## 21.1 Swap Space

The first thing we will need to do is to reserve some space on the disk for moving pages back and forth. In operating systems, we generally refer to such space as **swap space**, because we *swap* pages out of memory to it and *swap* pages into memory from it. Thus, we will simply assume that the OS can read from and write to the swap space, in page-sized units. To do so, the OS will need to remember the **disk address** of a given page.

The size of the swap space is important, as ultimately it determines the maximum number of memory pages that can be in use by a system at a given time. Let us assume for simplicity that it is *very* large for now.

In the tiny example (Figure 21.1), you can see a little example of a 4-page physical memory and an 8-page swap space. In the example, three processes (Proc 0, Proc 1, and Proc 2) are actively sharing physical memory; each of the three, however, only have some of their valid pages in memory, with the rest located in swap space on disk. A fourth process (Proc 3) has all of its pages swapped out to disk, and thus clearly isn't currently running. One block of swap remains free. Even from this tiny example, hopefully you can see how using swap space allows the system to pretend that memory is larger than it actually is.

We should note that swap space is not the only on-disk location for swapping traffic. For example, assume you are running a program binary (e.g., `ls`, or your own compiled `main` program). The code pages from this binary are initially found on disk, and when the program runs, they are loaded into memory (either all at once when the program starts execution,

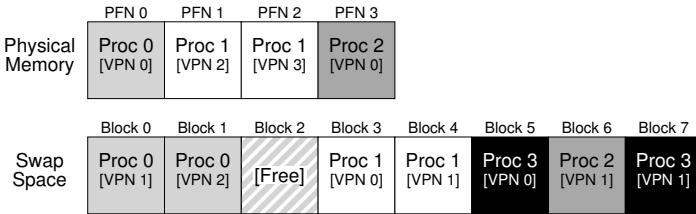


Figure 21.1: Physical Memory and Swap Space

or, as in modern systems, one page at a time when needed). However, if the system needs to make room in physical memory for other needs, it can safely re-use the memory space for these code pages, knowing that it can later swap them in again from the on-disk binary in the file system.

## 21.2 The Present Bit

Now that we have some space on the disk, we need to add some machinery higher up in the system in order to support swapping pages to and from the disk. Let us assume, for simplicity, that we have a system with a hardware-managed TLB.

Recall first what happens on a memory reference. The running process generates virtual memory references (for instruction fetches, or data accesses), and, in this case, the hardware translates them into physical addresses before fetching the desired data from memory.

Remember that the hardware first extracts the VPN from the virtual address, checks the TLB for a match (a **TLB hit**), and if a hit, produces the resulting physical address and fetches it from memory. This is hopefully the common case, as it is fast (requiring no additional memory accesses).

If the VPN is not found in the TLB (i.e., a **TLB miss**), the hardware locates the page table in memory (using the **page table base register**) and looks up the **page table entry (PTE)** for this page using the VPN as an index. If the page is valid and present in physical memory, the hardware extracts the PFN from the PTE, installs it in the TLB, and retries the instruction, this time generating a TLB hit; so far, so good.

If we wish to allow pages to be swapped to disk, however, we must add even more machinery. Specifically, when the hardware looks in the PTE, it may find that the page is *not present* in physical memory. The way the hardware (or the OS, in a software-managed TLB approach) determines this is through a new piece of information in each page-table entry, known as the **present bit**. If the present bit is set to one, it means the page is present in physical memory and everything proceeds as above; if it is set to zero, the page is *not* in memory but rather on disk somewhere. The act of accessing a page that is not in physical memory is commonly referred to as a **page fault**.

**ASIDE: SWAPPING TERMINOLOGY AND OTHER THINGS**

Terminology in virtual memory systems can be a little confusing and variable across machines and operating systems. For example, a **page fault** more generally could refer to any reference to a page table that generates a fault of some kind: this could include the type of fault we are discussing here, i.e., a page-not-present fault, but sometimes can refer to illegal memory accesses. Indeed, it is odd that we call what is definitely a legal access (to a page mapped into the virtual address space of a process, but simply not in physical memory at the time) a “fault” at all; really, it should be called a **page miss**. But often, when people say a program is “page faulting”, they mean that it is accessing parts of its virtual address space that the OS has swapped out to disk.

We suspect the reason that this behavior became known as a “fault” relates to the machinery in the operating system to handle it. When something unusual happens, i.e., when something the hardware doesn’t know how to handle occurs, the hardware simply transfers control to the OS, hoping it can make things better. In this case, a page that a process wants to access is missing from memory; the hardware does the only thing it can, which is raise an exception, and the OS takes over from there. As this is identical to what happens when a process does something illegal, it is perhaps not surprising that we term the activity a “fault.”

Upon a page fault, the OS is invoked to service the page fault. A particular piece of code, known as a **page-fault handler**, runs, and must service the page fault, as we now describe.

## 21.3 The Page Fault

Recall that with TLB misses, we have two types of systems: hardware-managed TLBs (where the hardware looks in the page table to find the desired translation) and software-managed TLBs (where the OS does). In either type of system, if a page is not present, the OS is put in charge to handle the page fault. The appropriately-named OS **page-fault handler** runs to determine what to do. Virtually all systems handle page faults in software; even with a hardware-managed TLB, the hardware trusts the OS to manage this important duty.

If a page is not present and has been swapped to disk, the OS will need to swap the page into memory in order to service the page fault. Thus, a question arises: how will the OS know where to find the desired page? In many systems, the page table is a natural place to store such information. Thus, the OS could use the bits in the PTE normally used for data such as the PFN of the page for a disk address. When the OS receives a page fault for a page, it looks in the PTE to find the address, and issues the request to disk to fetch the page into memory.

**ASIDE: WHY HARDWARE DOESN'T HANDLE PAGE FAULTS**

We know from our experience with the TLB that hardware designers are loathe to trust the OS to do much of anything. So why do they trust the OS to handle a page fault? There are a few main reasons. First, page faults to disk are *slow*; even if the OS takes a long time to handle a fault, executing tons of instructions, the disk operation itself is traditionally so slow that the extra overheads of running software are minimal. Second, to be able to handle a page fault, the hardware would have to understand swap space, how to issue I/Os to the disk, and a lot of other details which it currently doesn't know much about. Thus, for both reasons of performance and simplicity, the OS handles page faults, and even hardware types can be happy.

When the disk I/O completes, the OS will then update the page table to mark the page as present, update the PFN field of the page-table entry (PTE) to record the in-memory location of the newly-fetched page, and retry the instruction. This next attempt may generate a TLB miss, which would then be serviced and update the TLB with the translation (one could alternately update the TLB when servicing the page fault to avoid this step). Finally, a last restart would find the translation in the TLB and thus proceed to fetch the desired data or instruction from memory at the translated physical address.

Note that while the I/O is in flight, the process will be in the **blocked** state. Thus, the OS will be free to run other ready processes while the page fault is being serviced. Because I/O is expensive, this **overlap** of the I/O (page fault) of one process and the execution of another is yet another way a multiprogrammed system can make the most effective use of its hardware.

## 21.4 What If Memory Is Full?

In the process described above, you may notice that we assumed there is plenty of free memory in which to **page in** a page from swap space. Of course, this may not be the case; memory may be full (or close to it). Thus, the OS might like to first **page out** one or more pages to make room for the new page(s) the OS is about to bring in. The process of picking a page to kick out, or **replace** is known as the **page-replacement policy**.

As it turns out, a lot of thought has been put into creating a good page-replacement policy, as kicking out the wrong page can exact a great cost on program performance. Making the wrong decision can cause a program to run at disk-like speeds instead of memory-like speeds; in current technology that means a program could run 10,000 or 100,000 times slower. Thus, such a policy is something we should study in some detail; indeed, that is exactly what we will do in the next chapter. For now, it is good enough to understand that such a policy exists, built on top of the mechanisms described here.

```

1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)    // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10     else           // TLB Miss
11         PTEAddr = PTBR + (VPN * sizeof(PTE))
12         PTE = AccessMemory(PTEAddr)
13         if (PTE.Valid == False)
14             RaiseException(SEGMENTATION_FAULT)
15         else
16             if (CanAccess(PTE.ProtectBits) == False)
17                 RaiseException(PROTECTION_FAULT)
18             else if (PTE.Present == True)
19                 // assuming hardware-managed TLB
20                 TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
21                 RetryInstruction()
22             else if (PTE.Present == False)
23                 RaiseException(PAGE_FAULT)

```

Figure 21.2: **Page-Fault Control Flow Algorithm (Hardware)**

## 21.5 Page Fault Control Flow

With all of this knowledge in place, we can now roughly sketch the complete control flow of memory access. In other words, when somebody asks you “what happens when a program fetches some data from memory?”, you should have a pretty good idea of all the different possibilities. See the control flow in Figures 21.2 and 21.3 for more details; the first figure shows what the hardware does during translation, and the second what the OS does upon a page fault.

From the hardware control flow diagram in Figure 21.2, notice that there are now three important cases to understand when a TLB miss occurs. First, that the page was both **present** and **valid** (Lines 18–21); in this case, the TLB miss handler can simply grab the PFN from the PTE, retry the instruction (this time resulting in a TLB hit), and thus continue as described (many times) before. In the second case (Lines 22–23), the page fault handler must be run; although this was a legitimate page for the process to access (it is valid, after all), it is not present in physical memory. Third (and finally), the access could be to an invalid page, due for example to a bug in the program (Lines 13–14). In this case, no other bits in the PTE really matter; the hardware traps this invalid access, and the OS trap handler runs, likely terminating the offending process.

From the software control flow in Figure 21.3, we can see what the OS roughly must do in order to service the page fault. First, the OS must find a physical frame for the soon-to-be-faulted-in page to reside within; if there is no such page, we’ll have to wait for the replacement algorithm to run and kick some pages out of memory, thus freeing them for use here.

```

1 PFN = FindFreePhysicalPage()
2 if (PFN == -1)           // no free page found
3     PFN = EvictPage()    // run replacement algorithm
4 DiskRead(PTE.DiskAddr, pfn) // sleep (waiting for I/O)
5 PTE.present = True      // update page table with present
6 PTE.PFN    = PFN        // bit and translation (PFN)
7 RetryInstruction()       // retry instruction

```

Figure 21.3: Page-Fault Control Flow Algorithm (Software)

With a physical frame in hand, the handler then issues the I/O request to read in the page from swap space. Finally, when that slow operation completes, the OS updates the page table and retries the instruction. The retry will result in a TLB miss, and then, upon another retry, a TLB hit, at which point the hardware will be able to access the desired item.

## 21.6 When Replacements Really Occur

Thus far, the way we've described how replacements occur assumes that the OS waits until memory is entirely full, and only then replaces (evicts) a page to make room for some other page. As you can imagine, this is a little bit unrealistic, and there are many reasons for the OS to keep a small portion of memory free more proactively.

To keep a small amount of memory free, most operating systems thus have some kind of **high watermark** (*HW*) and **low watermark** (*LW*) to help decide when to start evicting pages from memory. How this works is as follows: when the OS notices that there are fewer than *LW* pages available, a background thread that is responsible for freeing memory runs. The thread evicts pages until there are *HW* pages available. The background thread, sometimes called the **swap daemon** or **page daemon**<sup>1</sup>, 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.

---

<sup>1</sup>The word "daemon", usually pronounced "demon", is an old term for a background thread or process that does something useful. Turns out (once again!) that the source of the term is Multics [CS94].

**TIP: DO WORK IN THE BACKGROUND**

When you have some work to do, it is often a good idea to do it in the **background** to increase efficiency and to allow for grouping of operations. Operating systems often do work in the background; for example, many systems buffer file writes in memory before actually writing the data to disk. Doing so has many possible benefits: increased disk efficiency, as the disk may now receive many writes at once and thus better be able to schedule them; improved latency of writes, as the application thinks the writes completed quite quickly; the possibility of work reduction, as the writes may need never to go to disk (i.e., if the file is deleted); and better use of **idle time**, as the background work may possibly be done when the system is otherwise idle, thus better utilizing the hardware [G+95].

## 21.7 Summary

In this brief chapter, we have introduced the notion of accessing more memory than is physically present within a system. To do so requires more complexity in page-table structures, as a **present bit** (of some kind) must be included to tell us whether the page is present in memory or not. When not, the operating system **page-fault handler** runs to service the **page fault**, and thus arranges for the transfer of the desired page from disk to memory, perhaps first replacing some pages in memory to make room for those soon to be swapped in.

Recall, importantly (and amazingly!), that these actions all take place **transparently** to the process. As far as the process is concerned, it is just accessing its own private, contiguous virtual memory. Behind the scenes, pages are placed in arbitrary (non-contiguous) locations in physical memory, and sometimes they are not even present in memory, requiring a fetch from disk. While we hope that in the common case a memory access is fast, in some cases it will take multiple disk operations to service it; something as simple as performing a single instruction can, in the worst case, take many milliseconds to complete.

## References

[CS94] "Take Our Word For It"

F. Corbato and R. Steinberg

Available: <http://www.takeourword.com/TOW146/page4.html>

Richard Steinberg writes: "Someone has asked me the origin of the word *daemon* as it applies to computing. Best I can tell based on my research, the word was first used by people on your team at Project MAC using the IBM 7094 in 1963." Professor Corbato replies: "Our use of the word *daemon* was inspired by the Maxwell's daemon of physics and thermodynamics (my background is in physics). Maxwell's *daemon* was an imaginary agent which helped sort molecules of different speeds and worked tirelessly in the background. We fancifully began to use the word *daemon* to describe background processes which worked tirelessly to perform system chores."

[D97] "Before Memory Was Virtual"

Peter Denning

From *In the Beginning: Recollections of Software Pioneers*, Wiley, November 1997

An excellent historical piece by one of the pioneers of virtual memory and working sets.

[G+95] "Idleness is not sloth"

Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, John Wilkes

USENIX ATC '95, New Orleans, Louisiana

A fun and easy-to-read discussion of how idle time can be better used in systems, with lots of good examples.

[LL82] "Virtual Memory Management in the VAX/VMS Operating System"

Hank Levy and P. Lipman

IEEE Computer, Vol. 15, No. 3, March 1982

Not the first place where such clustering was used, but a clear and simple explanation of how such a mechanism works.

## Homework (Measurement)

This homework introduces you to a new tool, **vmstat**, and how it can be used to get a sense of what your computer is doing with regards to memory, CPU, and I/O usage (with a focus on memory and swapping).

Read the associated README and examine the code in `mem.c` before proceeding to the exercises and questions below.

### Questions

1. First, open two separate terminal connections to the *same* machine, so that you can easily run something in one window and the other.

Now, in one window, run `vmstat 1`, which shows statistics about machine usage every second. Read the man page, the associated README, and any other information you need so that you can understand its output. Leave this window running `vmstat` for the rest of the exercises below.

Now, we will run the program `mem.c` but with very little memory usage. This can be accomplished by typing `./mem 1` (which uses only 1 MB of memory). How do the CPU usage statistics change when running `mem`? Do the numbers in the `user` time column make sense? How does this change when running more than one instance of `mem` at once?

2. Let's now start looking at some of the memory statistics while running `mem`. We'll focus on two columns: `swpd` (the amount of virtual memory used) and `free` (the amount of idle memory). Run `./mem 1024` (which allocates 1024 MB) and watch how these values change. Then kill the running program (by typing control-c) and watch again how the values change. What do you notice about the values? In particular, how does the `free` column change when the program exits? Does the amount of free memory increase by the expected amount when `mem` exits?
3. We'll next look at the `swap` columns (`si` and `so`), which indicate how much swapping is taking place to and from the disk. Of course, to activate these, you'll need to run `mem` with large amounts of memory. First, examine how much free memory is on your Linux system (for example, by typing `cat /proc/meminfo`; type `man proc` for details on the `/proc` file system and the types of information you can find there). One of the first entries in `/proc/meminfo` is the total amount of memory in your system. Let's assume it's something like 8 GB of memory; if so, start by running `mem 4000` (about 4 GB) and watching the swap in/out columns. Do they ever give non-zero values? Then, try with `5000`, `6000`, etc. What happens to these values as the program enters the second loop (and beyond), as compared to the first loop? How much data (total)

are swapped in and out during the second, third, and subsequent loops? (do the numbers make sense?)

4. Do the same experiments as above, but now watch the other statistics (such as CPU utilization, and block I/O statistics). How do they change when `mem` is running?
5. Now let's examine performance. Pick an input for `mem` that comfortably fits in memory (say 4000 if the amount of memory on the system is 8 GB). How long does loop 0 take (and subsequent loops 1, 2, etc.)? Now pick a size comfortably beyond the size of memory (say 12000 again assuming 8 GB of memory). How long do the loops take here? How do the bandwidth numbers compare? How different is performance when constantly swapping versus fitting everything comfortably in memory? Can you make a graph, with the size of memory used by `mem` on the x-axis, and the bandwidth of accessing said memory on the y-axis? Finally, how does the performance of the first loop compare to that of subsequent loops, for both the case where everything fits in memory and where it doesn't?
6. Swap space isn't infinite. You can use the tool `swapon` with the `-s` flag to see how much swap space is available. What happens if you try to run `mem` with increasingly large values, beyond what seems to be available in swap? At what point does the memory allocation fail?
7. Finally, if you're advanced, you can configure your system to use different swap devices using `swapon` and `swapoff`. Read the man pages for details. If you have access to different hardware, see how the performance of swapping changes when swapping to a classic hard drive, a flash-based SSD, and even a RAID array. How much can swapping performance be improved via newer devices? How close can you get to in-memory performance?

## Beyond Physical Memory: Policies

In a virtual memory manager, life is easy when you have a lot of free memory. A page fault occurs, you find a free page on the free-page list, and assign it to the faulting page. Hey, Operating System, congratulations! You did it again.

Unfortunately, things get a little more interesting when little memory is free. In such a case, this **memory pressure** forces the OS to start **paging out** pages to make room for actively-used pages. Deciding which page (or pages) to **evict** is encapsulated within the **replacement policy** of the OS; historically, it was one of the most important decisions the early virtual memory systems made, as older systems had little physical memory. Minimally, it is an interesting set of policies worth knowing a little more about. And thus our problem:

### THE CRUX: HOW TO DECIDE WHICH PAGE TO EVICT

How can the OS decide which page (or pages) to evict from memory? This decision is made by the replacement policy of the system, which usually follows some general principles (discussed below) but also includes certain tweaks to avoid corner-case behaviors.

### 22.1 Cache Management

Before diving into policies, we first describe the problem we are trying to solve in more detail. Given that main memory holds some subset of all the pages in the system, it can rightly be viewed as a **cache** for virtual memory pages in the system. Thus, our goal in picking a replacement policy for this cache is to minimize the number of **cache misses**, i.e., to minimize the number of times that we have to fetch a page from disk. Alternately, one can view our goal as maximizing the number of **cache hits**, i.e., the number of times a page that is accessed is found in memory.

Knowing the number of cache hits and misses let us calculate the **average memory access time (AMAT)** for a program (a metric computer

architects compute for hardware caches [HP06]). Specifically, given these values, we can compute the AMAT of a program as follows:

$$AMAT = (P_{Hit} \cdot T_M) + (P_{Miss} \cdot T_D) \quad (22.1)$$

where  $T_M$  represents the cost of accessing memory,  $T_D$  the cost of accessing disk,  $P_{Hit}$  the probability of finding the data item in the cache (a hit), and  $P_{Miss}$  the probability of not finding the data in the cache (a miss).  $P_{Hit}$  and  $P_{Miss}$  each vary from 0.0 to 1.0, and  $P_{Miss} + P_{Hit} = 1.0$ .

For example, let us imagine a machine with a (tiny) address space: 4KB, with 256-byte pages. Thus, a virtual address has two components: a 4-bit VPN (the most-significant bits) and an 8-bit offset (the least-significant bits). Thus, a process in this example can access  $2^4$  or 16 total virtual pages. In this example, the process generates the following memory references (i.e., virtual addresses): 0x000, 0x100, 0x200, 0x300, 0x400, 0x500, 0x600, 0x700, 0x800, 0x900. These virtual addresses refer to the first byte of each of the first ten pages of the address space (the page number being the first hex digit of each virtual address).

Let us further assume that every page except virtual page 3 is already in memory. Thus, our sequence of memory references will encounter the following behavior: hit, hit, hit, miss, hit, hit, hit, hit, hit. We can compute the **hit rate** (the percent of references found in memory): 90% ( $P_{Hit} = 0.9$ ), as 9 out of 10 references are in memory. The **miss rate** is obviously 10% ( $P_{Miss} = 0.1$ ).

To calculate AMAT, we simply need to know the cost of accessing memory and the cost of accessing disk. Assuming the cost of accessing memory ( $T_M$ ) is around 100 nanoseconds, and the cost of accessing disk ( $T_D$ ) is about 10 milliseconds, we have the following AMAT:  $0.9 \cdot 100\text{ns} + 0.1 \cdot 10\text{ms}$ , which is  $90\text{ns} + 1\text{ms}$ , or 1.00009 ms, or about 1 millisecond. If our hit rate had instead been 99.9%, the result is quite different: AMAT is 10.1 microseconds, or roughly 100 times faster. As the hit rate approaches 100%, AMAT approaches 100 nanoseconds.

Unfortunately, as you can see in this example, the cost of disk access is so high in modern systems that even a tiny miss rate will quickly dominate the overall AMAT of running programs. Clearly, we need to avoid as many misses as possible or run slowly, at the rate of the disk. One way to help with this is to carefully develop a smart policy, as we now do.

## 22.2 The Optimal Replacement Policy

To better understand how a particular replacement policy works, it would be nice to compare it to the best possible replacement policy. As it turns out, such an **optimal** policy was developed by Belady many years ago [B66] (he originally called it MIN). The optimal replacement policy leads to the fewest number of misses overall. Belady showed that a simple (but, unfortunately, difficult to implement!) approach that replaces the page that will be accessed *furthest in the future* is the optimal policy, resulting in the fewest-possible cache misses.

**TIP: COMPARING AGAINST OPTIMAL IS USEFUL**

Although optimal is not very practical as a real policy, it is incredibly useful as a comparison point in simulation or other studies. Saying that your fancy new algorithm has a 80% hit rate isn't meaningful in isolation; saying that optimal achieves an 82% hit rate (and thus your new approach is quite close to optimal) makes the result more meaningful and gives it context. Thus, in any study you perform, knowing what the optimal is lets you perform a better comparison, showing how much improvement is still possible, and also when you can *stop* making your policy better, because it is close enough to the ideal [AD03].

Hopefully, the intuition behind the optimal policy makes sense. Think about it like this: if you have to throw out some page, why not throw out the one that is needed the furthest from now? By doing so, you are essentially saying that all the other pages in the cache are more important than the one furthest out. The reason this is true is simple: you will refer to the other pages before you refer to the one furthest out.

Let's trace through a simple example to understand the decisions the optimal policy makes. Assume a program accesses the following stream of virtual pages: 0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1. Figure 22.1 shows the behavior of optimal, assuming a cache that fits three pages.

In the figure, you can see the following actions. Not surprisingly, the first three accesses are misses, as the cache begins in an empty state; such a miss is sometimes referred to as a **cold-start miss** (or **compulsory miss**). Then we refer again to pages 0 and 1, which both hit in the cache. Finally, we reach another miss (to page 3), but this time the cache is full; a replacement must take place! Which begs the question: which page should we replace? With the optimal policy, we examine the future for each page currently in the cache (0, 1, and 2), and see that 0 is accessed almost immediately, 1 is accessed a little later, and 2 is accessed furthest in the future. Thus the optimal policy has an easy choice: evict page 2, resulting in pages 0, 1, and 3 in the cache. The next three references are hits, but then

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

Figure 22.1: Tracing The Optimal Policy

#### ASIDE: TYPES OF CACHE MISSES

In the computer architecture world, architects sometimes find it useful to characterize misses by type, into one of three categories: compulsory, capacity, and conflict misses, sometimes called the **Three C's** [H87]. A **compulsory miss** (or **cold-start miss** [EF78]) occurs because the cache is empty to begin with and this is the first reference to the item; in contrast, a **capacity miss** occurs because the cache ran out of space and had to evict an item to bring a new item into the cache. The third type of miss (a **conflict miss**) arises in hardware because of limits on where an item can be placed in a hardware cache, due to something known as **set-associativity**; it does not arise in the OS page cache because such caches are always **fully-associative**, i.e., there are no restrictions on where in memory a page can be placed. See H&P for details [HP06].

we get to page 2, which we evicted long ago, and suffer another miss. Here the optimal policy again examines the future for each page in the cache (0, 1, and 3), and sees that as long as it doesn't evict page 1 (which is about to be accessed), we'll be OK. The example shows page 3 getting evicted, although 0 would have been a fine choice too. Finally, we hit on page 1 and the trace completes.

We can also calculate the hit rate for the cache: with 6 hits and 5 misses, the hit rate is  $\frac{\text{Hits}}{\text{Hits} + \text{Misses}}$  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<sup>1</sup>. Thus, in developing a real, deployable policy, we will focus on approaches that find some other way to decide which page to evict. The optimal policy will thus serve only as a comparison point, to know how close we are to "perfect".

### 22.3 A Simple Policy: FIFO

Many early systems avoided the complexity of trying to approach optimal and employed very simple replacement policies. For example, some systems used **FIFO** (first-in, first-out) replacement, where pages were simply placed in a queue when they enter the system; when a replacement occurs, the page on the tail of the queue (the "first-in" page) is evicted. FIFO has one great strength: it is quite simple to implement.

Let's examine how FIFO does on our example reference stream (Figure 22.2, page 5). We again begin our trace with three compulsory misses to pages 0, 1, and 2, and then hit on both 0 and 1. Next, page 3 is referenced, causing a miss; the replacement decision is easy with FIFO: pick the page

---

<sup>1</sup>If you can, let us know! We can become rich together. Or, like the scientists who "discovered" cold fusion, widely scorned and mocked [FP89].

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		First-in→ 0
1	Miss		First-in→ 0, 1
2	Miss		First-in→ 0, 1, 2
0	Hit		First-in→ 0, 1, 2
1	Hit		First-in→ 0, 1, 2
3	Miss	0	First-in→ 1, 2, 3
0	Miss	1	First-in→ 2, 3, 0
3	Hit		First-in→ 2, 3, 0
1	Miss	2	First-in→ 3, 0, 1
2	Miss	3	First-in→ 0, 1, 2
1	Hit		First-in→ 0, 1, 2

Figure 22.2: Tracing The FIFO Policy

that was the “first one” in (the cache state in the figure is kept in FIFO order, with the first-in page on the left), which is page 0. Unfortunately, our next access is to page 0, causing another miss and replacement (of page 1). We then hit on page 3, but miss on 1 and 2, and finally hit on 3.

Comparing FIFO to optimal, FIFO does notably worse: a 36.4% hit rate (or 57.1% excluding compulsory misses). FIFO simply can’t determine the importance of blocks: even though page 0 had been accessed a number of times, FIFO still kicks it out, simply because it was the first one brought into memory.

#### ASIDE: BELADY’S ANOMALY

Belady (of the optimal policy) and colleagues found an interesting reference stream that behaved a little unexpectedly [BNS69]. The memory-reference stream: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. The replacement policy they were studying was FIFO. The interesting part: how the cache hit rate changed when moving from a cache size of 3 to 4 pages.

In general, you would expect the cache hit rate to *increase* (get better) when the cache gets larger. But in this case, with FIFO, it gets worse! Calculate the hits and misses yourself and see. This odd behavior is generally referred to as **Belady’s Anomaly** (to the chagrin of his co-authors).

Some other policies, such as LRU, don’t suffer from this problem. Can you guess why? As it turns out, LRU has what is known as a **stack property** [M+70]. For algorithms with this property, a cache of size  $N + 1$  naturally includes the contents of a cache of size  $N$ . Thus, when increasing the cache size, hit rate will either stay the same or improve. FIFO and Random (among others) clearly do not obey the stack property, and thus are susceptible to anomalous behavior.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	0	1, 2, 3
0	Miss	1	2, 3, 0
3	Hit		2, 3, 0
1	Miss	3	2, 0, 1
2	Hit		2, 0, 1
1	Hit		2, 0, 1

Figure 22.3: Tracing The Random Policy

## 22.4 Another Simple Policy: Random

Another similar replacement policy is Random, which simply picks a random page to replace under memory pressure. Random has properties similar to FIFO; it is simple to implement, but it doesn't really try to be too intelligent in picking which blocks to evict. Let's look at how Random does on our famous example reference stream (see Figure 22.3).

Of course, how Random does depends entirely upon how lucky (or unlucky) Random gets in its choices. In the example above, Random does a little better than FIFO, and a little worse than optimal. In fact, we can run the Random experiment thousands of times and determine how it does in general. Figure 22.4 shows how many hits Random achieves over 10,000 trials, each with a different random seed. As you can see, sometimes (just over 40% of the time), Random is as good as optimal, achieving 6 hits on the example trace; sometimes it does much worse, achieving 2 hits or fewer. How Random does depends on the luck of the draw.

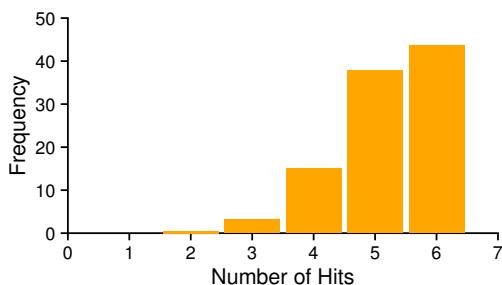


Figure 22.4: Random Performance Over 10,000 Trials

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2, 0, 1
3	Miss	2	LRU→ 0, 1, 3
0	Hit		LRU→ 1, 3, 0
3	Hit		LRU→ 1, 0, 3
1	Hit		LRU→ 0, 3, 1
2	Miss	0	LRU→ 3, 1, 2
1	Hit		LRU→ 3, 2, 1

Figure 22.5: Tracing The LRU Policy

## 22.5 Using History: LRU

Unfortunately, any policy as simple as FIFO or Random is likely to have a common problem: it might kick out an important page, one that is about to be referenced again. FIFO kicks out the page that was first brought in; if this happens to be a page with important code or data structures upon it, it gets thrown out anyhow, even though it will soon be paged back in. Thus, FIFO, Random, and similar policies are not likely to approach optimal; something smarter is needed.

As we did with scheduling policy, to improve our guess at the future, we once again lean on the past and use *history* as our guide. For example, if a program has accessed a page in the near past, it is likely to access it again in the near future.

One type of historical information a page-replacement policy could use is **frequency**; if a page has been accessed many times, perhaps it should not be replaced as it clearly has some value. A more commonly-used property of a page is its **recency** of access; the more recently a page has been accessed, perhaps the more likely it will be accessed again.

This family of policies is based on what people refer to as the **principle of locality** [D70], which basically is just an observation about programs and their behavior. What this principle says, quite simply, is that programs tend to access certain code sequences (e.g., in a loop) and data structures (e.g., an array accessed by the loop) quite frequently; we should thus try to use history to figure out which pages are important, and keep those pages in memory when it comes to eviction time.

And thus, a family of simple historically-based algorithms are born. The **Least-Frequently-Used** (LFU) policy replaces the least-frequently-used page when an eviction must take place. Similarly, the **Least-Recently-Used** (LRU) policy replaces the least-recently-used page. These algorithms are easy to remember: once you know the name, you know exactly what it does, which is an excellent property for a name.

To better understand LRU, let's examine how LRU does on our exam-

## ASIDE: TYPES OF LOCALITY

There are two types of locality that programs tend to exhibit. The first is known as **spatial locality**, which states that if a page  $P$  is accessed, it is likely the pages around it (say  $P - 1$  or  $P + 1$ ) will also likely be accessed. The second is **temporal locality**, which states that pages that have been accessed in the near past are likely to be accessed again in the near future. The assumption of the presence of these types of locality plays a large role in the caching hierarchies of hardware systems, which deploy many levels of instruction, data, and address-translation caching to help programs run fast when such locality exists.

Of course, the **principle of locality**, as it is often called, is no hard-and-fast rule that all programs must obey. Indeed, some programs access memory (or disk) in rather random fashion and don't exhibit much or any locality in their access streams. Thus, while locality is a good thing to keep in mind while designing caches of any kind (hardware or software), it does not *guarantee* success. Rather, it is a heuristic that often proves useful in the design of computer systems.

ple reference stream. Figure 22.5 (page 7) shows the results. From the figure, you can see how LRU can use history to do better than stateless policies such as Random or FIFO. In the example, LRU evicts page 2 when it first has to replace a page, because 0 and 1 have been accessed more recently. It then replaces page 0 because 1 and 3 have been accessed more recently. In both cases, LRU's decision, based on history, turns out to be correct, and the next references are thus hits. Thus, in our simple example, LRU does as well as possible, matching optimal in its performance<sup>2</sup>.

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.

---

<sup>2</sup>OK, we cooked the results. But sometimes cooking is necessary to prove a point.

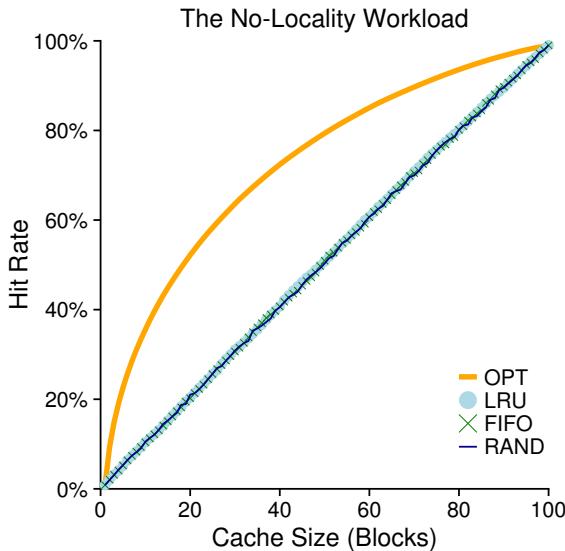


Figure 22.6: The No-Locality Workload

Figure 22.6 plots the results of the experiment for optimal, LRU, Random, and FIFO. The y-axis of the figure shows the hit rate that each policy achieves; the x-axis varies the cache size as described above.

We can draw a number of conclusions from the graph. First, when there is no locality in the workload, it doesn't matter much which realistic policy you are using; LRU, FIFO, and Random all perform the same, with the hit rate exactly determined by the size of the cache. Second, when the cache is large enough to fit the entire workload, it also doesn't matter which policy you use; all policies (even Random) converge to a 100% hit rate when all the referenced blocks fit in cache. Finally, you can see that optimal performs noticeably better than the realistic policies; peeking into the future, if it were possible, does a much better job of replacement.

The next workload we examine is called the "80-20" workload, which exhibits locality: 80% of the references are made to 20% of the pages (the "hot" pages); the remaining 20% of the references are made to the remaining 80% of the pages (the "cold" pages). In our workload, there are a total 100 unique pages again; thus, "hot" pages are referred to most of the time, and "cold" pages the remainder. Figure 22.7 (page 10) shows how the policies perform with this workload.

As you can see from the figure, while both random and FIFO do reasonably well, LRU does better, as it is more likely to hold onto the hot pages; as those pages have been referred to frequently in the past, they are likely to be referred to again in the near future. Optimal once again does better, showing that LRU's historical information is not perfect.

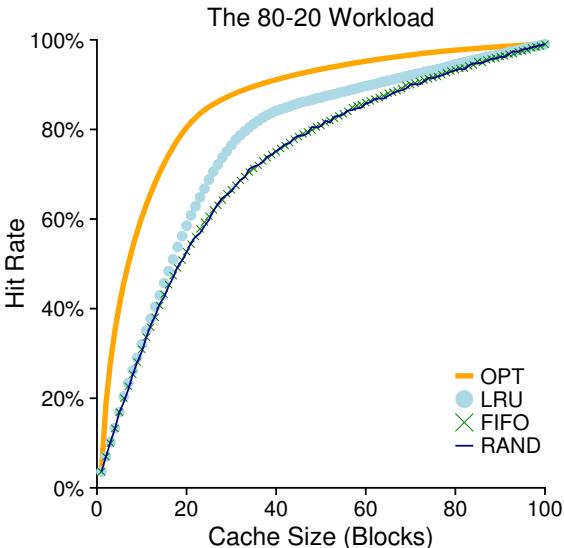


Figure 22.7: The 80-20 Workload

You might now be wondering: is LRU’s improvement over Random and FIFO really that big of a deal? The answer, as usual, is “it depends.” If each miss is very costly (not uncommon), then even a small increase in hit rate (reduction in miss rate) can make a huge difference on performance. If misses are not so costly, then of course the benefits possible with LRU are not nearly as important.

Let’s look at one final workload. We call this one the “looping sequential” workload, as in it, we refer to 50 pages in sequence, starting at 0, then 1, ..., up to page 49, and then we loop, repeating those accesses, for a total of 10,000 accesses to 50 unique pages. The last graph in Figure 22.8 shows the behavior of the policies under this workload.

This workload, common in many applications (including important commercial applications such as databases [CD85]), represents a worst-case for both LRU and FIFO. These algorithms, under a looping-sequential workload, kick out older pages; unfortunately, due to the looping nature of the workload, these older pages are going to be accessed sooner than the pages that the policies prefer to keep in cache. Indeed, even with a cache of size 49, a looping-sequential workload of 50 pages results in a 0% hit rate. Interestingly, Random fares notably better, not quite approaching optimal, but at least achieving a non-zero hit rate. Turns out that random has some nice properties; one such property is not having weird corner-case behaviors.

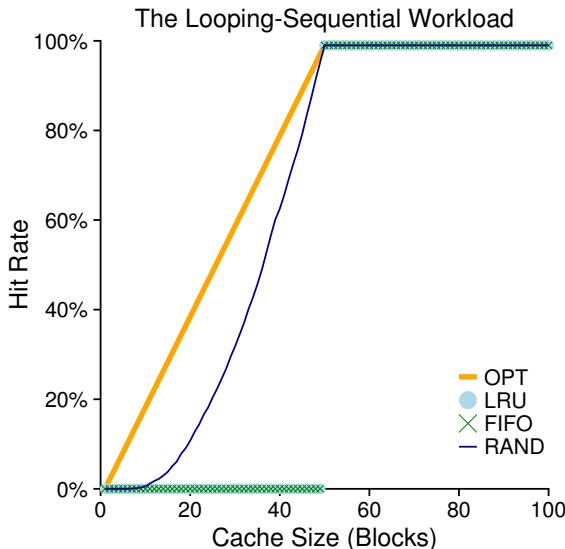


Figure 22.8: The Looping Workload

## 22.7 Implementing Historical Algorithms

As you can see, an algorithm such as LRU can generally do a better job than simpler policies like FIFO or Random, which may throw out important pages. Unfortunately, historical policies present us with a new challenge: how do we implement them?

Let's take, for example, LRU. To implement it perfectly, we need to do a lot of work. Specifically, upon each *page access* (i.e., each memory access, whether an instruction fetch or a load or store), we must update some data structure to move this page to the front of the list (i.e., the MRU side). Contrast this to FIFO, where the FIFO list of pages is only accessed when a page is *evicted* (by removing the first-in page) or when a new page is added to the list (to the last-in side). To keep track of which pages have been least- and most-recently used, the system has to do some accounting work *on every memory reference*. Clearly, without great care, such accounting could greatly reduce performance.

One method that could help speed this up is to add a little bit of hardware support. For example, a machine could update, on each page access, a time field in memory (for example, this could be in the per-process page table, or just in some separate array in memory, with one entry per physical page of the system). Thus, when a page is accessed, the time field would be set, by hardware, to the current time. Then, when replacing a page, the OS could simply scan all the time fields in the system to find the least-recently-used page.

Unfortunately, as the number of pages in a system grows, scanning a huge array of times just to find the absolute least-recently-used page is prohibitively expensive. Imagine a modern machine with 4GB of memory, chopped into 4KB pages. This machine has 1 million pages, and thus finding the LRU page will take a long time, even at modern CPU speeds. Which begs the question: do we really need to find the absolute oldest page to replace? Can we instead survive with an approximation?

#### CRUX: HOW TO IMPLEMENT AN LRU REPLACEMENT POLICY

Given that it will be expensive to implement perfect LRU, can we approximate it in some way, and still obtain the desired behavior?

## 22.8 Approximating LRU

As it turns out, the answer is yes: approximating LRU is more feasible from a computational-overhead standpoint, and indeed it is what many modern systems do. The idea requires some hardware support, in the form of a **use bit** (sometimes called the **reference bit**), the first of which was implemented in the first system with paging, the Atlas one-level store [KE+62]. There is one use bit per page of the system, and the use bits live in memory somewhere (they could be in the per-process page tables, for example, or just in an array somewhere). Whenever a page is referenced (i.e., read or written), the use bit is set by hardware to 1. The hardware never clears the bit, though (i.e., sets it to 0); that is the responsibility of the OS.

How does the OS employ the use bit to approximate LRU? Well, there could be a lot of ways, but with the **clock algorithm** [C69], one simple approach was suggested. Imagine all the pages of the system arranged in a circular list. A **clock hand** points to some particular page to begin with (it doesn't really matter which). When a replacement must occur, the OS checks if the currently-pointed to page  $P$  has a use bit of 1 or 0. If 1, this implies that page  $P$  was recently used and thus is *not* a good candidate for replacement. Thus, the use bit for  $P$  set to 0 (cleared), and the clock hand is incremented to the next page ( $P + 1$ ). The algorithm continues until it finds a use bit that is set to 0, implying this page has not been recently used (or, in the worst case, that all pages have been and that we have now searched through the entire set of pages, clearing all the bits).

Note that this approach is not the only way to employ a use bit to approximate LRU. Indeed, any approach which periodically clears the use bits and then differentiates between which pages have use bits of 1 versus 0 to decide which to replace would be fine. The clock algorithm of Corbató'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.

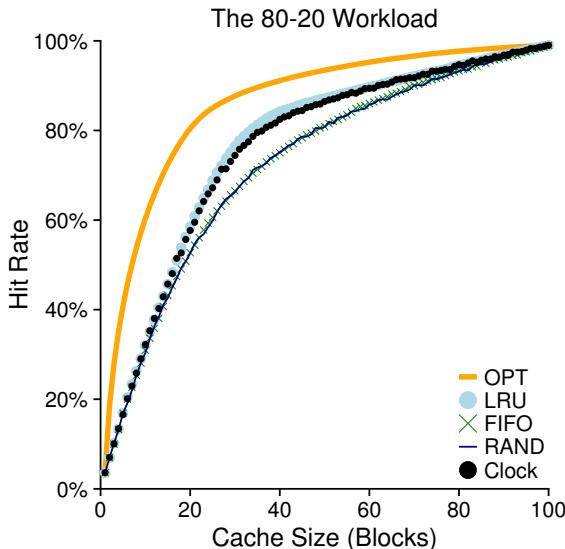


Figure 22.9: The 80-20 Workload With Clock

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 Corbató [C69]) that is commonly made is the additional consideration of whether a page has been modified or not while in memory. The reason for this: if a page has been **modified** and is thus **dirty**, it must be written back to disk to evict it, which is expensive. If it has not been modified (and is thus **clean**), the eviction is free; the physical frame can simply be reused for other purposes without additional I/O. Thus, some VM systems prefer to evict clean pages over dirty pages.

To support this behavior, the hardware should include a **modified bit** (a.k.a. **dirty bit**). This bit is set any time a page is written, and thus can be incorporated into the page-replacement algorithm. The clock algorithm, for example, could be changed to scan for pages that are both unused and clean to evict first; failing to find those, then for unused pages that are dirty, and so forth.

## 22.10 Other VM Policies

Page replacement is not the only policy the VM subsystem employs (though it may be the most important). For example, the OS also has to decide *when* to bring a page into memory. This policy, sometimes called the **page selection** policy (as it was called by Denning [D70]), presents the OS with some different options.

For most pages, the OS simply uses **demand paging**, which means the OS brings the page into memory when it is accessed, “on demand” as it were. Of course, the OS could guess that a page is about to be used, and thus bring it in ahead of time; this behavior is known as **prefetching** and should only be done when there is reasonable chance of success. For example, some systems will assume that if a code page  $P$  is brought into memory, that code page  $P+1$  will likely soon be accessed and thus should be brought into memory too.

Another policy determines how the OS writes pages out to disk. Of course, they could simply be written out one at a time; however, many systems instead collect a number of pending writes together in memory and write them to disk in one (more efficient) write. This behavior is usually called **clustering** or simply **grouping** of writes, and is effective because of the nature of disk drives, which perform a single large write more efficiently than many small ones.

## 22.11 Thrashing

Before closing, we address one final question: what should the OS do when memory is simply oversubscribed, and the memory demands of the set of running processes simply exceeds the available physical memory? In this case, the system will constantly be paging, a condition sometimes referred to as **thrashing** [D70].

Some earlier operating systems had a fairly sophisticated set of mechanisms to both detect and cope with thrashing when it took place. For example, given a set of processes, a system could decide not to run a subset of processes, with the hope that the reduced set of processes’ **working sets** (the pages that they are using actively) fit in memory and thus can make progress. This approach, generally known as **admission control**, states that it is sometimes better to do less work well than to try to do everything at once poorly, a situation we often encounter in real life as well as in modern computer systems (sadly).

Some current systems take more a draconian approach to memory overload. For example, some versions of Linux run an **out-of-memory killer** when memory is oversubscribed; this daemon chooses a memory-intensive process and kills it, thus reducing memory in a none-too-subtle manner. While successful at reducing memory pressure, this approach can have problems, if, for example, it kills the X server and thus renders any applications requiring the display unusable.

## 22.12 Summary

We have seen the introduction of a number of page-replacement (and other) policies, which are part of the VM subsystem of all modern operating systems. Modern systems add some tweaks to straightforward LRU approximations like clock; for example, **scan resistance** is an important part of many modern algorithms, such as ARC [MM03]. Scan-resistant algorithms are usually LRU-like but also try to avoid the worst-case behavior of LRU, which we saw with the looping-sequential workload. Thus, the evolution of page-replacement algorithms continues.

However, in many cases the importance of said algorithms has decreased, as the discrepancy between memory-access and disk-access times has increased. Because paging to disk is so expensive, the cost of frequent paging is prohibitive. Thus, the best solution to excessive paging is often a simple (if intellectually dissatisfying) one: buy more memory.

## References

- [AD03] "Run-Time Adaptation in River"  
 Remzi H. Arpacı-Dusseau  
 ACM TOCS, 21:1, February 2003  
*A summary of one of the authors' dissertation work on a system named River. Certainly one place where he learned that comparison against the ideal is an important technique for system designers.*
- [B66] "A Study of Replacement Algorithms for Virtual-Storage Computer"  
 Laszlo A. Belady  
 IBM Systems Journal 5(2): 78-101, 1966  
*The paper that introduces the simple way to compute the optimal behavior of a policy (the MIN algorithm).*
- [BNS69] "An Anomaly in Space-time Characteristics of Certain Programs Running in a Paging Machine"  
 L. A. Belady and R. A. Nelson and G. S. Shedler  
 Communications of the ACM, 12:6, June 1969  
*Introduction of the little sequence of memory references known as Belady's Anomaly. How do Nelson and Shedler feel about this name, we wonder?*
- [CD85] "An Evaluation of Buffer Management Strategies for Relational Database Systems"  
 Hong-Tai Chou and David J. DeWitt  
 VLDB '85, Stockholm, Sweden, August 1985  
*A famous database paper on the different buffering strategies you should use under a number of common database access patterns. The more general lesson: if you know something about a workload, you can tailor policies to do better than the general-purpose ones usually found in the OS.*
- [C69] "A Paging Experiment with the Multics System"  
 F.J. Corbato  
 Included in a Festschrift published in honor of Prof. P.M. Morse  
 MIT Press, Cambridge, MA, 1969  
*The original (and hard to find!) reference to the clock algorithm, though not the first usage of a use bit. Thanks to H. Balakrishnan of MIT for digging up this paper for us.*
- [D70] "Virtual Memory"  
 Peter J. Denning  
 Computing Surveys, Vol. 2, No. 3, September 1970  
*Denning's early and famous survey on virtual memory systems.*
- [EF78] "Cold-start vs. Warm-start Miss Ratios"  
 Malcolm C. Easton and Ronald Fagin  
 Communications of the ACM, 21:10, October 1978  
*A good discussion of cold-start vs. warm-start misses.*
- [FP89] "Electrochemically Induced Nuclear Fusion of Deuterium"  
 Martin Fleischmann and Stanley Pons  
 Journal of Electroanalytical Chemistry, Volume 26, Number 2, Part 1, April, 1989  
*The famous paper that would have revolutionized the world in providing an easy way to generate nearly-infinite power from jars of water with a little metal in them. Unfortunately, the results published (and widely publicized) by Pons and Fleischmann turned out to be impossible to reproduce, and thus these two well-meaning scientists were discredited (and certainly, mocked). The only guy really happy about this result was Marvin Hawkins, whose name was left off this paper even though he participated in the work; he thus avoided having his name associated with one of the biggest scientific goofs of the 20th century.*

[HP06] "Computer Architecture: A Quantitative Approach"

John Hennessy and David Patterson

Morgan-Kaufmann, 2006

*A great and marvelous book about computer architecture. Read it!*

[H87] "Aspects of Cache Memory and Instruction Buffer Performance"

Mark D. Hill

Ph.D. Dissertation, U.C. Berkeley, 1987

*Mark Hill, in his dissertation work, introduced the Three C's, which later gained wide popularity with its inclusion in H&P [HP06]. The quote from therein: "I have found it useful to partition misses ... into three components intuitively based on the cause of the misses (page 49)."*

[KE+62] "One-level Storage System"

T. Kilburn, and D.B.G. Edwards and M.J. Lanigan and F.H. Sumner

IRE Trans. EC-11:2, 1962

*Although Atlas had a use bit, it only had a very small number of pages, and thus the scanning of the use bits in large memories was not a problem the authors solved.*

[M+70] "Evaluation Techniques for Storage Hierarchies"

R. L. Mattson, J. Gecsei, D. R. Slutz, I. L. Traiger

IBM Systems Journal, Volume 9:2, 1970

*A paper that is mostly about how to simulate cache hierarchies efficiently; certainly a classic in that regard, as well for its excellent discussion of some of the properties of various replacement algorithms. Can you figure out why the stack property might be useful for simulating a lot of different-sized caches at once?*

[MM03] "ARC: A Self-Tuning, Low Overhead Replacement Cache"

Nimrod Megiddo and Dharmendra S. Modha

FAST 2003, February 2003, San Jose, California

*An excellent modern paper about replacement algorithms, which includes a new policy, ARC, that is now used in some systems. Recognized in 2014 as a "Test of Time" award winner by the storage systems community at the FAST '14 conference.*

## Homework

This simulator, `paging-policy.py`, allows you to play around with different page-replacement policies. See the README for details.

## Questions

1. Generate random addresses with the following arguments: `-s 0 -n 10`, `-s 1 -n 10`, and `-s 2 -n 10`. Change the policy from FIFO, to LRU, to OPT. Compute whether each access in said address traces are hits or misses.
2. For a cache of size 5, generate worst-case address reference streams for each of the following policies: FIFO, LRU, and MRU (worst-case reference streams cause the most misses possible. For the worst case reference streams, how much bigger of a cache is needed to improve performance dramatically and approach OPT?)
3. Generate a random trace (use python or perl). How would you expect the different policies to perform on such a trace?
4. Now generate a trace with some locality. How can you generate such a trace? How does LRU perform on it? How much better than RAND is LRU? How does CLOCK do? How about CLOCK with different numbers of clock bits?
5. Use a program like valgrind to instrument a real application and generate a virtual page reference stream. For example, running `valgrind --tool=lackey --trace-mem=yes ls` will output a nearly-complete reference trace of every instruction and data reference made by the program `ls`. To make this useful for the simulator above, you'll have to first transform each virtual memory reference into a virtual page-number reference (done by masking off the offset and shifting the resulting bits downward). How big of a cache is needed for your application trace in order to satisfy a large fraction of requests? Plot a graph of its working set as the size of the cache increases.

# Working Sets

**Annual income twenty pounds, annual expenditure nineteen [pounds], nineteen [shillings] and six [pence] ... result happiness.**

**Annual income twenty pounds, annual expenditure twenty pounds ought and six ... result misery.**

*Wilkins Micawber (Charles Dickens)*

## LRU is not enough

The success of LRU (Least Recently Used) replacement algorithms is the stuff of legends. Nobody knows what tomorrow will bring, but for most purposes our best guess is that the near future will be like the recent past. Unless you have inside information (i.e. references are not random, and you know understand the pattern) it is generally held that there is little profit to be gained from trying to out-smart LRU. The reason LRU works so well is that most programs exhibit some degree of temporal and spatial locality:

- if they use some code or data, they are likely to come back to access the same locations again.
- If they access code or data in a particular page, they are likely to reference other code (or data) in the same vicinity.

But these are statements about a single program or process. If we are doing Round-Robin time-sharing with multiple processes we give each process a brief opportunity to run, after which we run all the other processes before running it again. With the exception of shared text segments, separate processes seldom access the same pages.

- the most recently used pages in memory belong to the process that will not be run for a long time.
- the least recently used pages in memory belong to the process that is just about to run again.
- this destroys strict temporal and spatial locality, as reference behavior becomes periodic (rather than continuous).

In the absence of temporal and spatial locality, Global LRU will make poor decisions. Global LRU and Round-Robin scheduling are great algorithms, but they do not work well as a team. It might make sense to give each process its own dedicated set of page frames. When that process needed a new page, we would LRU replace the oldest page in that set. This would be *per-process* LRU, and that might work very well with Round-Robin scheduling.

## The concept of a Working Set

As we discovered in our discussion of paging:

- We do not need to give each process as many pages of physical memory as appear in the virtual address space.
- We do not even need to give each process as many pages of its virtual address as it will ever access.

It is a good thing if a process experiences occasional page faults, and has to replace old (no longer interesting) pages with new ones. What we want to avoid is page faults due to running a process in too little memory. We want to keep the page-faults down to a manageable rate. The ideal mean-time-between-page-faults would be equal to the time-slice length. As we give a process fewer and fewer page frames in which to operate, the page fault rate rises and our performance gets worse (as we spend less time doing useful computation and more time processing page faults).

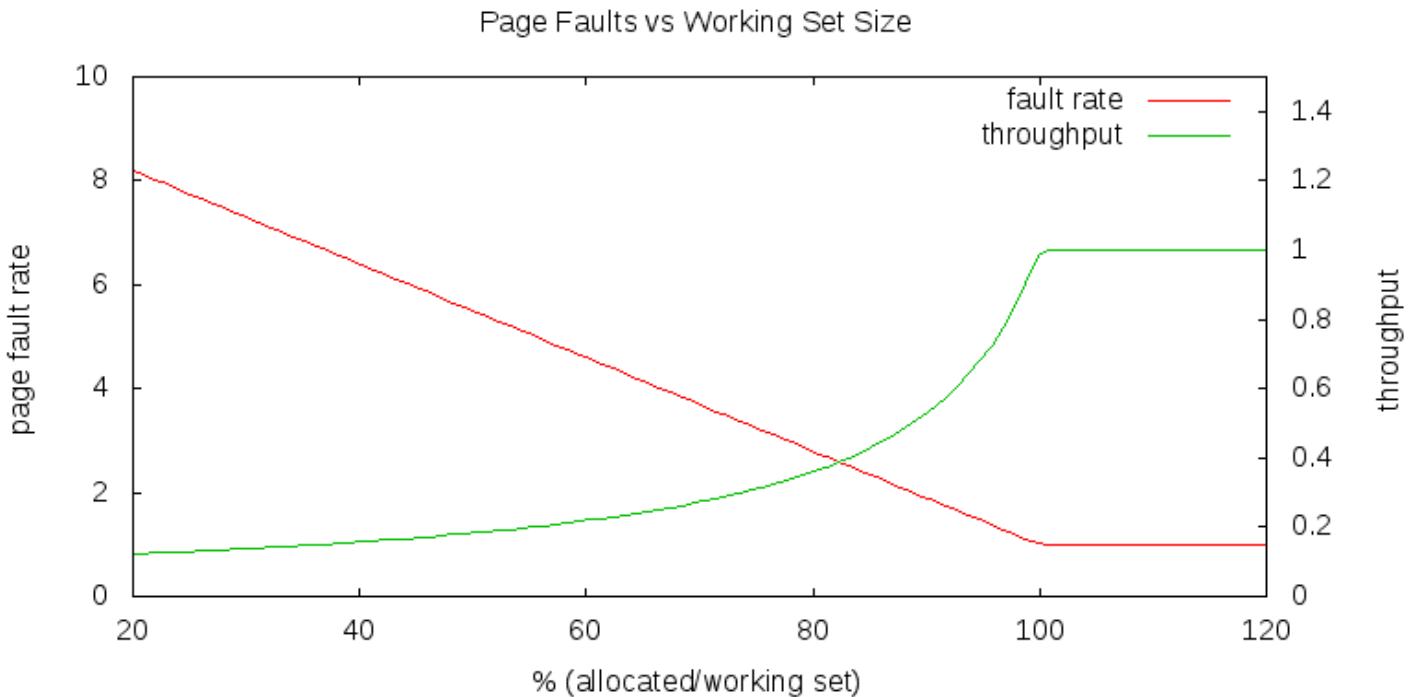
The dramatic degradation in system performance associated with not having enough memory to efficiently run all of the ready processes is called *thrashing*.

- If we think that we can have 25 processes in the ready queue, then we had better have enough memory for all 25 of those processes.
- If we only have enough memory to run 15 of those processes, then we should take the other 10 out of the ready queue (e.g. by swapping them out).
- The improved performance resulting from reducing the number of page faults will more than make up for the delays processes experience while being swapped between primary and secondary storage.

There is, for any given computation, at any given time, a number of pages such that:

- if we increase the number of page frames allocated to that process, it makes very little difference in the performance.
- if we reduce the number of page frames allocated to that process, the performance suffers noticeably.

We will call that number the process' *working set size* (at that particular time).



## How large is a working set

Different computations require different amounts of memory. A tight loop operating on a few variables might need only two pages. Complex combinations of functions applied to large amounts of data could require hundreds of thousands of pages. Not only are different programs likely to require different amounts of memory, a single program may require different amounts of memory at different points during its execution.

Requiring more memory is not necessarily a bad thing ... it merely reflects the costs of that particular computation. When we explored scheduling we saw that different processes have different degrees of interactivity, and that if we could characterize the interactivity of each process we could more efficiently schedule it. The same is true of working set size. We can infer a process' working set size by observing its behavior. If a process is experiencing many page faults, its working set is larger than the memory currently allocated to it. If a process is not experiencing page faults, it may have too much memory allocated to it. If we can allocate the right amount of memory to each process, we will minimize our page faults (overhead) and maximize our throughput (efficiency).

## Implementing Working Set replacement

Regularly scanning all of memory to identify the least recently used page is a very expensive process. With global LRU, we saw that a clock-like scan was a very inexpensive way to achieve a very similar affect. The clock hand position was a surrogate for age:

- the most recently examined pages are immediately behind the hand.
- the pages we examined longest ago are immediately in front of the hand.
- if the page immediately in front of the hand has not been referenced since the last time it has been scanned, it must be very old ... even if it is not actually the oldest page in memory.

We can use a very similar approach to implement working sets. But we will need to maintain a little bit more information:

- each page frame is associated with an *owning process*
- each process has an *accumulated CPU time*

- each page frame will have a *last referenced* time, value, taken from the *accumulated CPU* timer of its *owning process*.
- we maintain a *target age* parameter ... which is *keep in memory* goal for all pages.

The new scan algorithm will be:

```

while ...
{
    // see if this page is old enough to replace
    owningProc = page->owner;
    if (page.referenced) {
        // assume it was just referenced
        page.lastRef = owningProc->accumulatedTime;
        page.referenced = 0;
    } else {
        // has it gone unreferenced long enough?
        age = owningProc->accumulatedTime - page.lastRef;
        if (age > targetAge)
            return(page);
    }
}

```

The key elements of this algorithm are:

- Age decisions are not made on the basis of clock time, but accumulated CPU time in the owning process. Pages only age when their owner runs without referencing them.
- If we find a page that has been referenced since the last scan, we assume it was just referenced.
- If a page is younger than the target age, we do not want to replace it ... since recycling of young pages indicates we may be thrashing.
- If a page is older than the target age, we take it away from its current owner, and give it to a new (needy) process.
- If there are no pages older than the target age, we probably have too many processes to fit in the available memory, and we need to start swapping some of them out.

## Dynamic Equilibrium to the rescue

This is often referred to as a *page stealing* algorithm, because a process that needs another page *steals* it from a process that does not seem to need it as much.

- Every process is continuously losing pages that it has not recently referenced.
- Every process is continuously stealing pages from other processes.
- Processes that reference more pages more often, will accumulate larger working sets.
- Processes that reference fewer pages less often will find their working sets reduced.
- When programs change their behavior, their allocated working sets adjust promptly and automatically.

This is a dynamic equilibrium mechanism. The continuously opposing processes of stealing and being stolen from will automatically allocate the available memory to the running processes in proportion to their working set sizes. It does not try to manage processes to any pre-configured notion of a reasonable working set size. Rather it manages memory to minimize the number of page faults, and to avoid thrashing.

# Inter-Process Communication

## Introduction

We can divide process interactions into a two broad categories:

1. the coordination of operations with other processes:
  - synchronization (e.g. mutexes and condition variables)
  - the exchange of signals (e.g. *kill(2)*)
  - control operations (e.g. *fork(2)*, *wait(2)*, *ptrace(2)*)
2. the exchange of data between processes:
  - uni-directional data processing pipelines
  - bi-directional interactions

The first of these are discussed in other readings and lectures. This is an introduction to the exchange of data between processes.

## Simple Uni-Directional Byte Streams

These are easy to create and trivial to use. A pipe can be opened by a parent and inherited by a child, who simply reads standard input and writes standard output. Such pipelines can be used as part of a standard processing model:

```
macro-processor | compiler | assembler > output
```

or as custom constructions for one-off tasks:

```
find . -newer $TIMESTAMP | grep -v '*.o' | tar cfz archive.tgz -T -
```

All such uses have a few key characteristics in common:

- Each program accepts a byte-stream input, and produces a byte-stream output that is a well defined function of the input.
- Each program in the pipe-line operates independently, and is unaware that the others exist or what they might do.
- Byte streams are inherently unstructured. If there is any structure to the data they carry (e.g. newline-delimited lines or comma-separated-values) these conventions are implemented by parsers in the affected applications.
- Ensuring that the output of one program is suitable input to the next is the responsibility of the agent who creates the pipeline.
- Similar results could be obtained by writing the output of each program into a (temporary) file, and then using that file as input to the next program.

Pipes are temporary files with a few special features, that recognize the difference between a file (whose contents are relatively static) and an inter-process data stream:

- If the reader exhausts all of the data in the pipe, but there is still an open write file descriptor, the reader does not get an *End of File*(EOF). Rather the reader is blocked until more data becomes available, or the write side is closed.
- The available buffering capacity of the pipe may be limited. If the writer gets too far ahead of the reader, the operating system may block the writer until the reader catches up. This is called *flow control*.
- Writing to a pipe that no longer has an open read file descriptor is illegal, and the writer will be sent a  exception signal (*SIGPIPE*).

- When the read and write file descriptors are both closed, the file is automatically deleted.

Because a pipeline (in principle) represents a closed system, the only data privacy mechanisms tend to be the protections on the initial input files and final output files. There is generally no authentication or encryption of the data being passed between successive processes in the pipeline.

## Named Pipes and Mail-Boxes

A named-pipe (*fifo(7)*) is a baby-step towards explicit connections. It can be thought of as a persistent pipe, whose reader and writer(s) can open it by name, rather than inheriting it from a *pipe(2)* system call. A normal pipe is custom-plumbed to interconnect processes started by a single user. A named pipe can be used as a rendezvous point for unrelated processes. Named pipes are almost as simple to use as ordinary pipes, but ...

- Readers and writers have no way of authenticating one-another's identities.
- Writes from multiple writers may be interspersed, with no indications of which bytes came from whom.
- They do not enable clean fail-overs from a failed reader to its successor.
- All readers and writers must be running on the same node.

Recognizing these limitations, some operating systems have created more general inter-process communication mechanisms, often called *mailboxes*. While implementations differ, common features include:

- Data is not a byte-stream. Rather each write is stored and delivered as a distinct message.
- Each write is accompanied by authenticated identification information about its sender.
- Unprocessed messages remain in the mailbox after the death of a reader and can be retrieved by the next reader.

But mailboxes still subject to single node/single operating system restrictions, and most distributed applications are now based on general and widely standardized network protocols.

## General Network Connections

Most operating systems now provide a fairly standard set of network communications APIs. The associated Linux APIs are:

- *socket(2)* ... create an inter-process communication end-point with an associated protocol and data model.
- *bind(2)* ... associate a *socket* with a local network address.
- *connect(2)* ... establish a connection to a remote network address.
- *listen(2)* ... await an incoming connection request.
- *accept(2)* ... accept an incoming connection request.
- *send(2)* ... send a message over a socket.
- *recv(2)* ... receive a message from a socket.

These APIs directly provide a range of different communications options:

- byte streams over reliable connections (e.g. TCP)
- best effort data-grams (e.g. UDP)

But they also form a foundation for higher level communication/service models. A few examples include:

- Remote Procedure Calls ... distributed request/response APIs.
- RESTful service models ... layered on top of HTTP GETs and PUTs.
- Publish/Subscribe services ... content based information flow.

Using more general networking models enables processes to interact with services all over the world, but this adds considerable complexity:

- Ensuring interoperability with software running under different operating systems on computers with different instruction set architectures.
- Dealing with the security issues associated with exchanging data and services with unknown systems over public networks.
- Discovering the addresses of (a constantly changing set of) servers.
- Detecting and recovering from (relatively common) connection and node failures.

Applications are forced to choose between a simple but strictly local model (pipes) or a general but highly complex model (network communications). But there is yet another issue: performance. Protocol stacks may be many layers deep, and data may be processed and copied many times. Network communication may have limited throughput, and high latencies.

## Shared Memory

Sometimes performance is more important than generality.

- The network drivers, MPEG decoders, and video rendering in a set top box are guaranteed to be local. Making these operations more efficient can greatly reduce the required processing power, resulting in a smaller form-factor, reduced heat dissipation, and a lower product cost.
- The network drivers, protocol interpreters, write-back cache, RAID implementation, and back-end drivers in a storage array are guaranteed to be local. Significantly reducing the execution time per write operation is the difference between an industry leader and road-kill.

High performance for Inter-Process Communication means generally means:

- efficiency ... low cost (instructions, nano-seconds, watts) per byte transferred.
- throughput ... maximum number of bytes (or messages) per second that can be transferred.
- latency ... minimum delay between sender write and receiver read.

If we want ultra high performance Inter-Process Communication between two local processes, buffering the data through the operating system and/or protocol stacks is not the way to get it. The fastest and most efficient way to move data between processes is through shared memory:

- create a file for communication.
- each process maps that file into its virtual address space.
- the shared segment might be locked-down, so that it is never paged out.
- the communicating processes agree on a set of data structures (e.g. polled lock-free circular buffers) in the shared segment.
- anything written into the shared memory segment will be immediately visible to all of the processes that have it mapped in to their address spaces.

Once the shared segment has been created and mapped into the participating process' address spaces, the Operating System plays no role in the subsequent data exchanges. Moving data in this way is extremely efficient and blindingly fast ... but (like all good things) this performance comes at a price:

- This can only be used between processes on the same memory bus.
- A bug in one of the processes can easily destroy the communications data structures.
- There is no authentication (beyond access control on the shared file) of which data came from which process.

## Network Connections and Out-of-Band Signals

In most cases, event completions can be reported simply by sending a message (announcing the completion) to the waiter. But what if there are megabytes of queued requests, and we want to send a message to abort those queued requests? Data sent down a network connection is FIFO ... and one of the problems with FIFO scheduling is the delays waiting for the processing of earlier but longer messages. Occasionally, we would like to make it possible for an important message to go directly to front of the line.

If the recipient was local, we might consider sending a signal that could invoke a registered handler, and flush (without processing) all of the buffered data. This works because the signals travel over a different channel than the buffered data. Such communication is often called *out-of-band*, because it does not travel over the normal data path.

We can achieve a similar effect with network based services by opening multiple communications channels:

- one heavily used channel for normal requests
- another reserved for out-of-band requests

The server on the far end periodically polls the out-of-band channel before taking requests from the normal communications channel. This adds a little overhead to the processing, but makes it possible to preempt queued operations. The chosen polling interval represents a trade-off between added overhead (to check for out-of-band messages) and how long we might go (how much wasted work we might do) before noticing an out-of-band message.



# Named Pipes (FIFOs - First In First Out)

## 6.3.1 Basic Concepts

A named pipe works much like a regular pipe, but does have some noticeable differences.

- Named pipes exist as a device special file in the file system.
- Processes of different ancestry can share data through a named pipe.
- When all I/O is done by sharing processes, the named pipe remains in the file system for later use.

## 6.3.2 Creating a FIFO

There are several ways of creating a named pipe. The first two can be done directly from the shell.

```
mknod MYFIFO p
mkfifo a=rw MYFIFO
```

The above two commands perform identical operations, with one exception. The mkfifo command provides a hook for altering the permissions on the FIFO file directly after creation. With mknod, a quick call to the chmod command will be necessary.

FIFO files can be quickly identified in a physical file system by the ``p'' indicator seen here in a long directory listing:

```
$ ls -l MYFIFO
prw-r--r--    1 root      root          0 Dec 14 22:15 MYFIFO|
```

Also notice the vertical bar (``pipe sign'') located directly after the file name. Another great reason to run Linux, eh?

To create a FIFO in C, we can make use of the mknod() system call:

---

```
LIBRARY FUNCTION: mknod();
```

```
PROTOTYPE: int mknod( char *pathname, mode_t mode, dev_t dev);
RETURNS: 0 on success,
         -1 on error: errno = EFAULT (pathname invalid)
                     EACCES (permission denied)
                     ENAMETOOLONG (pathname too long)
```

```
ENOENT (invalid pathname)
ENOTDIR (invalid pathname)
(see man page for mknod for others)
```

NOTES: Creates a filesystem node (file, device file, or FIFO)

---

I will leave a more detailed discussion of mknod() to the man page, but let's consider a simple example of FIFO creation from C:

```
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

In this case, the file ``/tmp/MYFIFO'' is created as a FIFO file. The requested permissions are ``0666'', although they are affected by the umask setting as follows:

```
final_umask = requested_permissions & ~original_umask
```

A common trick is to use the umask() system call to temporarily zap the umask value:

```
umask(0);
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

In addition, the third argument to mknod() is ignored unless we are creating a device file. In that instance, it should specify the major and minor numbers of the device file.

### 6.3.3 FIFO Operations

I/O operations on a FIFO are essentially the same as for normal pipes, with one major exception. An ``open'' system call or library function should be used to physically open up a channel to the pipe. With half-duplex pipes, this is unnecessary, since the pipe resides in the kernel and not on a physical filesystem. In our examples, we will treat the pipe as a stream, opening it up with fopen(), and closing it with fclose().

Consider a simple server process:

```
*****
*
Excerpt from "Linux Programmer's Guide - Chapter 6"
(C)opyright 1994-1995, Scott Burkett
*****
MODULE: fifoserver.c
*****
/
#include <stdio.h>
#include <stdlib.h>
```

```

#include <sys/stat.h>
#include <unistd.h>

#include <linux/stat.h>

#define FIFO_FILE      "MYFIFO"

int main(void)
{
    FILE *fp;
    char readbuf[80];

    /* Create the FIFO if it does not exist */
    umask(0);
    mknod(FIFO_FILE, S_IFIFO|0666, 0);

    while(1)
    {
        fp = fopen(FIFO_FILE, "r");
        fgets(readbuf, 80, fp);
        printf("Received string: %s\n", readbuf);
        fclose(fp);
    }

    return(0);
}

```

Since a FIFO blocks by default, run the server in the background after you compile it:

```
$ fifoserver&
```

We will discuss a FIFO's blocking action in a moment. First, consider the following simple client frontend to our server:

```

*****
*
Excerpt from "Linux Programmer's Guide - Chapter 6"
(C)opyright 1994-1995, Scott Burkett

*****
MODULE: fifoclient.c

*****
/
#include <stdio.h>
#include <stdlib.h>

#define FIFO_FILE      "MYFIFO"

int main(int argc, char *argv[])
{
    FILE *fp;

    if ( argc != 2 ) {

```

```

        printf("USAGE: fifoclient [string]\n");
        exit(1);
    }

    if((fp = fopen(FIFO_FILE, "w")) == NULL) {
        perror("fopen");
        exit(1);
    }

    fputs(argv[1], fp);

    fclose(fp);
    return(0);
}

```

### **6.3.4 Blocking Actions on a FIFO**

Normally, blocking occurs on a FIFO. In other words, if the FIFO is opened for reading, the process will "block" until some other process opens it for writing. This action works vice-versa as well. If this behavior is undesirable, the O\_NONBLOCK flag can be used in an open() call to disable the default blocking action.

In the case with our simple server, we just shoved it into the background, and let it do its blocking there. The alternative would be to jump to another virtual console and run the client end, switching back and forth to see the resulting action.

### **6.3.5 The Infamous SIGPIPE Signal**

On a last note, pipes must have a reader and a writer. If a process tries to write to a pipe that has no reader, it will be sent the SIGPIPE signal from the kernel. This is imperative when more than two processes are involved in a pipeline.

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [CONFORMING TO](#) |  
[NOTES](#) | [BUGS](#) | [EXAMPLE](#) | [SEE ALSO](#) | [COLOPHON](#)

  
 Search online pages
**SEND(2)****Linux Programmer's Manual****SEND(2)****NAME**[top](#)

`send`, `sendto`, `sendmsg` – send a message on a socket

**SYNOPSIS**[top](#)

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int sockfd, const void *buf, size_t len, int flags);

ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
              const struct sockaddr *dest_addr, socklen_t addrlen);

ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

**DESCRIPTION**[top](#)

The system calls `send()`, `sendto()`, and `sendmsg()` are used to transmit a message to another socket.

The `send()` call may be used only when the socket is in a *connected* state (so that the intended recipient is known). The only difference between `send()` and `write(2)` is the presence of *flags*. With a zero *flags* argument, `send()` is equivalent to `write(2)`. Also, the following call

```
send(sockfd, buf, len, flags);
```

is equivalent to

```
sendto(sockfd, buf, len, flags, NULL, 0);
```

The argument `sockfd` is the file descriptor of the sending socket.

If `sendto()` is used on a connection-mode (`SOCK_STREAM`, `SOCK_SEQPACKET`) socket, the arguments `dest_addr` and `addrlen` are ignored (and the error `EISCONN` may be returned when they are not `NULL` and `0`), and the error `ENOTCONN` is returned when the socket was not actually connected. Otherwise, the address of the target is given by `dest_addr` with `addrlen` specifying its size. For `sendmsg()`, the address of the target is given by `msg.msg_name`, with `msg.msg_namelen` specifying its size.

For **send()** and **sendto()**, the message is found in **buf** and has length **len**. For **sendmsg()**, the message is pointed to by the elements of the array **msg.msg iov**. The **sendmsg()** call also allows sending ancillary data (also known as control information).

If the message is too long to pass atomically through the underlying protocol, the error **EMSGSIZE** is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a **send()**. Locally detected errors are indicated by a return value of -1.

When the message does not fit into the send buffer of the socket, **send()** normally blocks, unless the socket has been placed in nonblocking I/O mode. In nonblocking mode it would fail with the error **EAGAIN** or **EWOULDBLOCK** in this case. The **select(2)** call may be used to determine when it is possible to send more data.

### The **flags** argument

The **flags** argument is the bitwise OR of zero or more of the following flags.

#### **MSG\_CONFIRM** (since Linux 2.3.15)

Tell the link layer that forward progress happened: you got a successful reply from the other side. If the link layer doesn't get this it will regularly reprobe the neighbor (e.g., via a unicast ARP). Only valid on **SOCK\_DGRAM** and **SOCK\_RAW** sockets and currently implemented only for IPv4 and IPv6. See [arp\(7\)](#) for details.

#### **MSG\_DONTROUTE**

Don't use a gateway to send out the packet, send to hosts only on directly connected networks. This is usually used only by diagnostic or routing programs. This is defined only for protocol families that route; packet sockets don't.

#### **MSG\_DONTWAIT** (since Linux 2.2)

Enables nonblocking operation; if the operation would block, **EAGAIN** or **EWOULDBLOCK** is returned. This provides similar behavior to setting the **O\_NONBLOCK** flag (via the **fcntl(2)** **F\_SETFL** operation), but differs in that **MSG\_DONTWAIT** is a per-call option, whereas **O\_NONBLOCK** is a setting on the open file description (see [open\(2\)](#)), which will affect all threads in the calling process and as well as other processes that hold file descriptors referring to the same open file description.

#### **MSG\_EOR** (since Linux 2.2)

Terminates a record (when this notion is supported, as for sockets of type **SOCK\_SEQPACKET**).

#### **MSG\_MORE** (since Linux 2.4.4)

The caller has more data to send. This flag is used with TCP sockets to obtain the same effect as the **TCP\_CORK** socket

option (see [tcp\(7\)](#)), with the difference that this flag can be set on a per-call basis.

Since Linux 2.6, this flag is also supported for UDP sockets, and informs the kernel to package all of the data sent in calls with this flag set into a single datagram which is transmitted only when a call is performed that does not specify this flag. (See also the **UDP\_CORK** socket option described in [udp\(7\)](#).)

#### **MSG\_NOSIGNAL** (since Linux 2.2)

Don't generate a **SIGPIPE** signal if the peer on a stream-oriented socket has closed the connection. The **EPIPE** error is still returned. This provides similar behavior to using [sigaction\(2\)](#) to ignore **SIGPIPE**, but, whereas **MSG\_NOSIGNAL** is a per-call feature, ignoring **SIGPIPE** sets a process attribute that affects all threads in the process.

#### **MSG\_OOB**

Sends *out-of-band* data on sockets that support this notion (e.g., of type **SOCK\_STREAM**); the underlying protocol must also support *out-of-band* data.

### **sendmsg()**

The definition of the **msghdr** structure employed by **sendmsg()** is as follows:

```
struct msghdr {
    void          *msg_name;      /* optional address */
    socklen_t      msg_namelen;   /* size of address */
    struct iovec  *msg_iov;       /* scatter/gather array */
    size_t         msg iovlen;    /* # elements in msg_iov */
    void          *msg_control;   /* ancillary data, see below */
    size_t         msg_controllen; /* ancillary data buffer len */
    int            msg_flags;     /* flags (unused) */
};
```

The **msg\_name** field is used on an unconnected socket to specify the target address for a datagram. It points to a buffer containing the address; the **msg\_namelen** field should be set to the size of the address. For a connected socket, these fields should be specified as NULL and 0, respectively.

The **msg\_iov** and **msg iovlen** fields specify scatter-gather locations, as for [writev\(2\)](#).

You may send control information using the **msg\_control** and **msg\_controllen** members. The maximum control buffer length the kernel can process is limited per socket by the value in **/proc/sys/net/core/optmem\_max**; see [socket\(7\)](#).

The **msg\_flags** field is ignored.

**RETURN VALUE**[top](#)

On success, these calls return the number of bytes sent. On error, -1 is returned, and [\*errno\*](#) is set appropriately.

**ERRORS**[top](#)

These are some standard errors generated by the socket layer. Additional errors may be generated and returned from the underlying protocol modules; see their respective manual pages.

**EACCES** (For UNIX domain sockets, which are identified by pathname)  
Write permission is denied on the destination socket file, or search permission is denied for one of the directories the path prefix. (See [\*path\\_resolution\(7\)\*](#).)

(For UDP sockets) An attempt was made to send to a network/broadcast address as though it was a unicast address.

**EAGAIN** or **EWOULDBLOCK**

The socket is marked nonblocking and the requested operation would block. POSIX.1-2001 allows either error to be returned for this case, and does not require these constants to have the same value, so a portable application should check for both possibilities.

**EAGAIN** (Internet domain datagram sockets) The socket referred to by *sockfd* had not previously been bound to an address and, upon attempting to bind it to an ephemeral port, it was determined that all port numbers in the ephemeral port range are currently in use. See the discussion of [\*/proc/sys/net/ipv4/ip\\_local\\_port\\_range\*](#) in [\*ip\(7\)\*](#).

**EBADF** *sockfd* is not a valid open file descriptor.

**ECONNRESET**

Connection reset by peer.

**EDESTADDRREQ**

The socket is not connection-mode, and no peer address is set.

**EFAULT** An invalid user space address was specified for an argument.

**EINTR** A signal occurred before any data was transmitted; see [\*signal\(7\)\*](#).

**EINVAL** Invalid argument passed.

**EISCONN**

The connection-mode socket was connected already but a recipient was specified. (Now either this error is returned, or the recipient specification is ignored.)

**EMSGSIZE**

The socket type requires that message be sent atomically, and the size of the message to be sent made this impossible.

**ENOBUFS**

The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion. (Normally, this does not occur in Linux. Packets are just silently dropped when a device queue overflows.)

**ENOMEM** No memory available.

**ENOTCONN**

The socket is not connected, and no target has been given.

**ENOTSOCK**

The file descriptor *sockfd* does not refer to a socket.

**EOPNOTSUPP**

Some bit in the *flags* argument is inappropriate for the socket type.

**EPIPE** The local end has been shut down on a connection oriented socket. In this case, the process will also receive a **SIGPIPE** unless **MSG\_NOSIGNAL** is set.

**CONFORMING TO**[top](#)

4.4BSD, SVr4, POSIX.1-2001. These interfaces first appeared in 4.2BSD.

POSIX.1-2001 describes only the **MSG\_OOB** and **MSG\_EOR** flags. POSIX.1-2008 adds a specification of **MSG\_NOSIGNAL**. The **MSG\_CONFIRM** flag is a Linux extension.

**NOTES**[top](#)

According to POSIX.1-2001, the *msg\_controllen* field of the *msghdr* structure should be typed as *socklen\_t*, but glibc currently types it as *size\_t*.

See [sendmmsg\(2\)](#) for information about a Linux-specific system call that can be used to transmit multiple datagrams in a single call.

**BUGS**[top](#)

Linux may return **EPIPE** instead of **ENOTCONN**.

**EXAMPLE**[top](#)

An example of the use of **sendto()** is shown in [getaddrinfo\(3\)](#).

**SEE ALSO**[top](#)

[fcntl\(2\)](#), [getsockopt\(2\)](#), [recv\(2\)](#), [select\(2\)](#), [sendfile\(2\)](#), [sendmmsg\(2\)](#), [shutdown\(2\)](#), [socket\(2\)](#), [write\(2\)](#), [cmsg\(3\)](#), [ip\(7\)](#), [socket\(7\)](#), [tcp\(7\)](#), [udp\(7\)](#)

**COLOPHON**[top](#)

This page is part of release 4.08 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at  
<https://www.kernel.org/doc/man-pages/>.

**Linux****2016-03-15****SEND(2)**

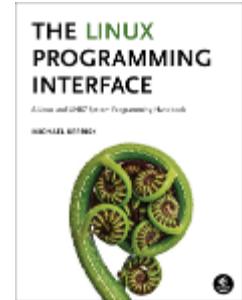
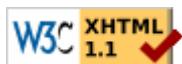
---

**Copyright and license for this manual page**

HTML rendering created 2016-10-08 by Michael Kerrisk, author of *The Linux Programming Interface*, maintainer of the Linux *man-pages* project.

For details of in-depth Linux/UNIX system programming training courses that I teach, look [here](#).

Hosting by [jambit GmbH](#).



[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [CONFORMING TO](#) |  
[NOTES](#) | [EXAMPLE](#) | [SEE ALSO](#) | [COLOPHON](#)

Search online pages

**RECV(2)****Linux Programmer's Manual****RECV(2)****NAME**[top](#)

`recv`, `recvfrom`, `recvmsg` – receive a message from a socket

**SYNOPSIS**[top](#)

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recv(int sockfd, void *buf, size_t len, int flags);

ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);

ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

**DESCRIPTION**[top](#)

The `recv()`, `recvfrom()`, and `recvmsg()` calls are used to receive messages from a socket. They may be used to receive data on both connectionless and connection-oriented sockets. This page first describes common features of all three system calls, and then describes the differences between the calls.

The only difference between `recv()` and `read(2)` is the presence of `flags`. With a zero `flags` argument, `recv()` is generally equivalent to `read(2)` (but see NOTES). Also, the following call

```
recv(sockfd, buf, len, flags);
```

is equivalent to

```
recvfrom(sockfd, buf, len, flags, NULL, NULL);
```

All three calls return the length of the message on successful completion. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from.

If no messages are available at the socket, the receive calls wait for a message to arrive, unless the socket is nonblocking (see `fcntl(2)`), in which case the value -1 is returned and the external variable `errno` is set to `EAGAIN` or `EWOULDBLOCK`. The receive calls normally return any data available, up to the requested amount,

rather than waiting for receipt of the full amount requested.

An application can use `select(2)`, `poll(2)`, or `epoll(7)` to determine when more data arrives on a socket.

### The `flags` argument

The `flags` argument is formed by ORing one or more of the following values:

#### **MSG\_CMSG\_CLOEXEC** (`recvmsg()` only; since Linux 2.6.23)

Set the close-on-exec flag for the file descriptor received via a UNIX domain file descriptor using the `SCM_RIGHTS` operation (described in [unix\(7\)](#)). This flag is useful for the same reasons as the `O_CLOEXEC` flag of [open\(2\)](#).

#### **MSG\_DONTWAIT** (since Linux 2.2)

Enables nonblocking operation; if the operation would block, the call fails with the error `EAGAIN` or `EWOULDBLOCK`. This provides similar behavior to setting the `O_NONBLOCK` flag (via the `fcntl(2)` `F_SETFL` operation), but differs in that `MSG_DONTWAIT` is a per-call option, whereas `O_NONBLOCK` is a setting on the open file description (see [open\(2\)](#)), which will affect all threads in the calling process and as well as other processes that hold file descriptors referring to the same open file description.

#### **MSG\_ERRQUEUE** (since Linux 2.2)

This flag specifies that queued errors should be received from the socket error queue. The error is passed in an ancillary message with a type dependent on the protocol (for IPv4 `IP_RECVERR`). The user should supply a buffer of sufficient size. See [cmsg\(3\)](#) and [ip\(7\)](#) for more information. The payload of the original packet that caused the error is passed as normal data via `msg_iovec`. The original destination address of the datagram that caused the error is supplied via `msg_name`.

For local errors, no address is passed (this can be checked with the `cmsg_len` member of the `cmsghdr`). For error receives, the `MSG_ERRQUEUE` is set in the `msghdr`. After an error has been passed, the pending socket error is regenerated based on the next queued error and will be passed on the next socket operation.

The error is supplied in a `sock_extended_err` structure:

```
#define SO_EE_ORIGIN_NONE      0
#define SO_EE_ORIGIN_LOCAL     1
#define SO_EE_ORIGIN_ICMP      2
#define SO_EE_ORIGIN_ICMP6     3

struct sock_extended_err
{
    uint32_t ee_errno;        /* error number */
    uint8_t  ee_origin;       /* where the error originated */
```

```

    uint8_t ee_type;      /* type */
    uint8_t ee_code;      /* code */
    uint8_t ee_pad;       /* padding */
    uint32_t ee_info;     /* additional information */
    uint32_t ee_data;     /* other data */
    /* More data may follow */
};

struct sockaddr *SO_EE_OFFENDER(struct sock_extended_err *);

```

`ee_errno` contains the `errno` number of the queued error. `ee_origin` is the origin code of where the error originated. The other fields are protocol-specific. The macro `SOCK_EE_OFFENDER` returns a pointer to the address of the network object where the error originated from given a pointer to the ancillary message. If this address is not known, the `sa_family` member of the `sockaddr` contains `AF_UNSPEC` and the other fields of the `sockaddr` are undefined. The payload of the packet that caused the error is passed as normal data.

For local errors, no address is passed (this can be checked with the `cmsg_len` member of the `cmsghdr`). For error receives, the `MSG_ERRQUEUE` is set in the `msghdr`. After an error has been passed, the pending socket error is regenerated based on the next queued error and will be passed on the next socket operation.

#### **MSG\_OOB**

This flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus this flag cannot be used with such protocols.

#### **MSG\_PEEK**

This flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data.

#### **MSG\_TRUNC** (since Linux 2.2)

For raw (`AF_PACKET`), Internet datagram (since Linux 2.4.27/2.6.8), netlink (since Linux 2.6.22), and UNIX datagram (since Linux 3.4) sockets: return the real length of the packet or datagram, even when it was longer than the passed buffer.

For use with Internet stream sockets, see [tcp\(7\)](#).

#### **MSG\_WAITALL** (since Linux 2.2)

This flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if a signal is caught, an error or disconnect occurs, or the next data to be received is of a different type than that returned. This flag has no effect for datagram sockets.

**recvfrom()**

**recvfrom()** places the received message into the buffer *buf*. The caller must specify the size of the buffer in *len*.

If *src\_addr* is not NULL, and the underlying protocol provides the source address of the message, that source address is placed in the buffer pointed to by *src\_addr*. In this case, *addrlen* is a value-result argument. Before the call, it should be initialized to the size of the buffer associated with *src\_addr*. Upon return, *addrlen* is updated to contain the actual size of the source address. The returned address is truncated if the buffer provided is too small; in this case, *addrlen* will return a value greater than was supplied to the call.

If the caller is not interested in the source address, *src\_addr* and *addrlen* should be specified as NULL.

**recv()**

The **recv()** call is normally used only on a *connected* socket (see [connect\(2\)](#)). It is equivalent to the call:

```
recvfrom(fd, buf, len, flags, NULL, 0);
```

**recvmsg()**

The **recvmsg()** call uses a *msghdr* structure to minimize the number of directly supplied arguments. This structure is defined as follows in [\*<sys/socket.h>\*](#):

```
struct iovec {                                     /* Scatter/gather array items */
    void *iov_base;                                /* Starting address */
    size_t iov_len;                                 /* Number of bytes to transfer */
};

struct msghdr {                                   
    void *msg_name;                                /* optional address */
    socklen_t msg_namelen;                           /* size of address */
    struct iovec *msg_iov;                            /* scatter/gather array */
    size_t msg_iovlen;                              /* # elements in msg_iov */
    void *msg_control;                             /* ancillary data, see below */
    size_t msg_controllen;                         /* ancillary data buffer len */
    int msg_flags;                                 /* flags on received message */
};
```

The *msg\_name* field points to a caller-allocated buffer that is used to return the source address if the socket is unconnected. The caller should set *msg\_namelen* to the size of this buffer before this call; upon return from a successful call, *msg\_namelen* will contain the length of the returned address. If the application does not need to know the source address, *msg\_name* can be specified as NULL.

The fields *msg\_iov* and *msg\_iovlen* describe scatter-gather locations, as discussed in [readv\(2\)](#).

The field *msg\_control*, which has length *msg\_controllen*, points to a

buffer for other protocol control-related messages or miscellaneous ancillary data. When `recvmsg()` is called, `msg_controllen` should contain the length of the available buffer in `msg_control`; upon return from a successful call it will contain the length of the control message sequence.

The messages are of the form:

```
struct cmsghdr {
    size_t cmsg_len;      /* Data byte count, including header
                           (type is socklen_t in POSIX) */
    int    cmsg_level;   /* Originating protocol */
    int    cmsg_type;    /* Protocol-specific type */
/* followed by
   unsigned char cmsg_data[ ]; */
};
```

Ancillary data should be accessed only by the macros defined in [cmsg\(3\)](#).

As an example, Linux uses this ancillary data mechanism to pass extended errors, IP options, or file descriptors over UNIX domain sockets.

The `msg_flags` field in the `msghdr` is set on return of `recvmsg()`. It can contain several flags:

#### **MSG\_EOR**

indicates end-of-record; the data returned completed a record (generally used with sockets of type `SOCK_SEQPACKET`).

#### **MSG\_TRUNC**

indicates that the trailing portion of a datagram was discarded because the datagram was larger than the buffer supplied.

#### **MSG\_CTRUNC**

indicates that some control data were discarded due to lack of space in the buffer for ancillary data.

#### **MSG\_OOB**

is returned to indicate that expedited or out-of-band data were received.

#### **MSG\_ERRQUEUE**

indicates that no data was received but an extended error from the socket error queue.

## **RETURN VALUE**

[top](#)

These calls return the number of bytes received, or -1 if an error occurred. In the event of an error, `errno` is set to indicate the error.

When a stream socket peer has performed an orderly shutdown, the return value will be 0 (the traditional "end-of-file" return).

Datagram sockets in various domains (e.g., the UNIX and Internet domains) permit zero-length datagrams. When such a datagram is received, the return value is 0.

The value 0 may also be returned if the requested number of bytes to receive from a stream socket was 0.

## ERRORS

[top](#)

These are some standard errors generated by the socket layer. Additional errors may be generated and returned from the underlying protocol modules; see their manual pages.

### **EAGAIN** or **EWOULDBLOCK**

The socket is marked nonblocking and the receive operation would block, or a receive timeout had been set and the timeout expired before data was received. POSIX.1 allows either error to be returned for this case, and does not require these constants to have the same value, so a portable application should check for both possibilities.

**EBADF** The argument `sockfd` is an invalid file descriptor.

### **ECONNREFUSED**

A remote host refused to allow the network connection (typically because it is not running the requested service).

**EFAULT** The receive buffer pointer(s) point outside the process's address space.

**EINTR** The receive was interrupted by delivery of a signal before any data were available; see [signal\(7\)](#).

**EINVAL** Invalid argument passed.

**ENOMEM** Could not allocate memory for `recvmsg()`.

### **ENOTCONN**

The socket is associated with a connection-oriented protocol and has not been connected (see [connect\(2\)](#) and [accept\(2\)](#)).

### **ENOTSOCK**

The file descriptor `sockfd` does not refer to a socket.

## CONFORMING TO

[top](#)

POSIX.1-2001, POSIX.1-2008, 4.4BSD (these interfaces first appeared in 4.2BSD).

POSIX.1 describes only the **MSG\_OOB**, **MSG\_PEEK**, and **MSG\_WAITALL** flags.

**NOTES**[top](#)

If a zero-length datagram is pending, `read(2)` and `recv()` with a `flags` argument of zero provide different behavior. In this circumstance, `read(2)` has no effect (the datagram remains pending), while `recv()` consumes the pending datagram.

The `socklen_t` type was invented by POSIX. See also [accept\(2\)](#).

According to POSIX.1, the `msg_controllen` field of the `msghdr` structure should be typed as `socklen_t`, but glibc currently types it as `size_t`.

See [recvmsg\(2\)](#) for information about a Linux-specific system call that can be used to receive multiple datagrams in a single call.

**EXAMPLE**[top](#)

An example of the use of `recvfrom()` is shown in [getaddrinfo\(3\)](#).

**SEE ALSO**[top](#)

[fcntl\(2\)](#), [getsockopt\(2\)](#), [read\(2\)](#), [recvmsg\(2\)](#), [select\(2\)](#),  
[shutdown\(2\)](#), [socket\(2\)](#), [cmsg\(3\)](#), [sockatmark\(3\)](#), [socket\(7\)](#)

**COLOPHON**[top](#)

This page is part of release 4.08 of the Linux `man-pages` project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at  
<https://www.kernel.org/doc/man-pages/>.

Linux

2016-07-17

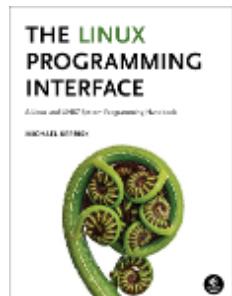
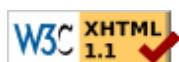
RECV(2)

[Copyright and license for this manual page](#)

HTML rendering created 2016-10-08 by Michael Kerrisk, author of [The Linux Programming Interface](#), maintainer of the Linux `man-pages` project.

For details of in-depth Linux/UNIX system programming training courses that I teach, look [here](#).

Hosting by [jambit GmbH](#).



[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) |  
[CONFORMING TO](#) | [AVAILABILITY](#) | [NOTES](#) | [BUGS](#) | [EXAMPLE](#) | [SEE ALSO](#) | [COLOPHON](#)

  
 Search online pages
**MMAP(2)****Linux Programmer's Manual****MMAP(2)****NAME**
[top](#)

`mmap`, `munmap` - map or unmap files or devices into memory

**SYNOPSIS**
[top](#)

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void *addr, size_t length);
```

See NOTES for information on feature test macro requirements.

**DESCRIPTION**
[top](#)

`mmap()` creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in `addr`. The `length` argument specifies the length of the mapping.

If `addr` is NULL, then the kernel chooses the address at which to create the mapping; this is the most portable method of creating a new mapping. If `addr` is not NULL, then the kernel takes it as a hint about where to place the mapping; on Linux, the mapping will be created at a nearby page boundary. The address of the new mapping is returned as the result of the call.

The contents of a file mapping (as opposed to an anonymous mapping; see `MAP_ANONYMOUS` below), are initialized using `length` bytes starting at offset `offset` in the file (or other object) referred to by the file descriptor `fd`. `offset` must be a multiple of the page size as returned by `sysconf(_SC_PAGE_SIZE)`.

The `prot` argument describes the desired memory protection of the mapping (and must not conflict with the open mode of the file). It is either `PROT_NONE` or the bitwise OR of one or more of the following flags:

**PROT\_EXEC** Pages may be executed.

**PROT\_READ** Pages may be read.

**PROT\_WRITE** Pages may be written.

**PROT\_NONE** Pages may not be accessed.

The *flags* argument determines whether updates to the mapping are visible to other processes mapping the same region, and whether updates are carried through to the underlying file. This behavior is determined by including exactly one of the following values in *flags*:

#### **MAP\_SHARED**

Share this mapping. Updates to the mapping are visible to other processes mapping the same region, and (in the case of file-backed mappings) are carried through to the underlying file. (To precisely control when updates are carried through to the underlying file requires the use of [msync\(2\)](#).)

#### **MAP\_PRIVATE**

Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file, and are not carried through to the underlying file. It is unspecified whether changes made to the file after the [mmap\(\)](#) call are visible in the mapped region.

Both of these flags are described in [POSIX.1-2001](#) and [POSIX.1-2008](#).

In addition, zero or more of the following values can be ORed in *flags*:

#### **MAP\_32BIT** (since Linux 2.4.20, 2.6)

Put the mapping into the first 2 Gigabytes of the process address space. This flag is supported only on x86-64, for 64-bit programs. It was added to allow thread stacks to be allocated somewhere in the first 2GB of memory, so as to improve context-switch performance on some early 64-bit processors. Modern x86-64 processors no longer have this performance problem, so use of this flag is not required on those systems. The **MAP\_32BIT** flag is ignored when **MAP\_FIXED** is set.

#### **MAP\_ANON**

Synonym for **MAP\_ANONYMOUS**. Deprecated.

#### **MAP\_ANONYMOUS**

The mapping is not backed by any file; its contents are initialized to zero. The *fd* argument is ignored; however, some implementations require *fd* to be -1 if **MAP\_ANONYMOUS** (or **MAP\_ANON**) is specified, and portable applications should ensure this. The *offset* argument should be zero. The use of **MAP\_ANONYMOUS** in conjunction with **MAP\_SHARED** is supported on Linux only since kernel 2.4.

#### **MAP\_DENYWRITE**

This flag is ignored. (Long ago, it signaled that attempts to write to the underlying file should fail with **ETXTBUSY**. But

this was a source of denial-of-service attacks.)

#### **MAP\_EXECUTABLE**

This flag is ignored.

#### **MAP\_FILE**

Compatibility flag. Ignored.

#### **MAP\_FIXED**

Don't interpret `addr` as a hint: place the mapping at exactly that address. `addr` must be a multiple of the page size. If the memory region specified by `addr` and `len` overlaps pages of any existing mapping(s), then the overlapped part of the existing mapping(s) will be discarded. If the specified address cannot be used, `mmap()` will fail. Because requiring a fixed address for a mapping is less portable, the use of this option is discouraged.

#### **MAP\_GROWSDOWN**

Used for stacks. Indicates to the kernel virtual memory system that the mapping should extend downward in memory.

#### **MAP\_HUGETLB** (since Linux 2.6.32)

Allocate the mapping using "huge pages." See the Linux kernel source file `Documentation/vm/hugetlbpage.txt` for further information, as well as NOTES, below.

#### **MAP\_HUGE\_2MB, MAP\_HUGE\_1GB** (since Linux 3.8)

Used in conjunction with **MAP\_HUGETLB** to select alternative hugetlb page sizes (respectively, 2 MB and 1 GB) on systems that support multiple hugetlb page sizes.

More generally, the desired huge page size can be configured by encoding the base-2 logarithm of the desired page size in the six bits at the offset **MAP\_HUGE\_SHIFT**. (A value of zero in this bit field provides the default huge page size; the default huge page size can be discovered via the `Hugepagesize` field exposed by `/proc/meminfo`.) Thus, the above two constants are defined as:

```
#define MAP_HUGE_2MB      (21 << MAP_HUGE_SHIFT)
#define MAP_HUGE_1GB       (30 << MAP_HUGE_SHIFT)
```

The range of huge page sizes that are supported by the system can be discovered by listing the subdirectories in `/sys/kernel/mm/hugepages`.

#### **MAP\_LOCKED** (since Linux 2.5.37)

Mark the mmaped region to be locked in the same way as `mlock(2)`. This implementation will try to populate (prefault) the whole range but the `mmap` call doesn't fail with **ENOMEM** if this fails. Therefore major faults might happen later on. So the semantic is not as strong as `mlock(2)`. One should use `mmap()` plus `mlock(2)` when major faults are not acceptable

after the initialization of the mapping. The **MAP\_LOCKED** flag is ignored in older kernels.

#### **MAP\_NONBLOCK** (since Linux 2.5.46)

Only meaningful in conjunction with **MAP\_POPULATE**. Don't perform read-ahead: create page tables entries only for pages that are already present in RAM. Since Linux 2.6.23, this flag causes **MAP\_POPULATE** to do nothing. One day, the combination of **MAP\_POPULATE** and **MAP\_NONBLOCK** may be reimplemented.

#### **MAP\_NORESERVE**

Do not reserve swap space for this mapping. When swap space is reserved, one has the guarantee that it is possible to modify the mapping. When swap space is not reserved one might get **SIGSEGV** upon a write if no physical memory is available. See also the discussion of the file [/proc/sys/vm/overcommit\\_memory](/proc/sys/vm/overcommit_memory) in [proc\(5\)](#). In kernels before 2.6, this flag had effect only for private writable mappings.

#### **MAP\_POPULATE** (since Linux 2.5.46)

Populate (prefault) page tables for a mapping. For a file mapping, this causes read-ahead on the file. This will help to reduce blocking on page faults later. **MAP\_POPULATE** is supported for private mappings only since Linux 2.6.23.

#### **MAP\_STACK** (since Linux 2.6.27)

Allocate the mapping at an address suitable for a process or thread stack. This flag is currently a no-op, but is used in the glibc threading implementation so that if some architectures require special treatment for stack allocations, support can later be transparently implemented for glibc.

#### **MAP\_UNINITIALIZED** (since Linux 2.6.33)

Don't clear anonymous pages. This flag is intended to improve performance on embedded devices. This flag is honored only if the kernel was configured with the **CONFIG\_MMAP\_ALLOW\_UNINITIALIZED** option. Because of the security implications, that option is normally enabled only on embedded devices (i.e., devices where one has complete control of the contents of user memory).

Of the above flags, only **MAP\_FIXED** is specified in POSIX.1-2001 and POSIX.1-2008. However, most systems also support **MAP\_ANONYMOUS** (or its synonym **MAP\_ANON**).

Some systems document the additional flags **MAP\_AUTOGROW**, **MAP\_AUTORESRV**, **MAP\_COPY**, and **MAP\_LOCAL**.

Memory mapped by **mmap()** is preserved across [fork\(2\)](#), with the same attributes.

A file is mapped in multiples of the page size. For a file that is not a multiple of the page size, the remaining memory is zeroed when

mapped, and writes to that region are not written out to the file. The effect of changing the size of the underlying file of a mapping on the pages that correspond to added or removed regions of the file is unspecified.

### **munmap()**

The **munmap()** system call deletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references. The region is also automatically unmapped when the process is terminated. On the other hand, closing the file descriptor does not unmap the region.

The address *addr* must be a multiple of the page size (but *length* need not be). All pages containing a part of the indicated range are unmapped, and subsequent references to these pages will generate **SIGSEGV**. It is not an error if the indicated range does not contain any mapped pages.

## RETURN VALUE

[top](#)

On success, **mmap()** returns a pointer to the mapped area. On error, the value **MAP\_FAILED** (that is, `(void *) -1`) is returned, and *errno* is set to indicate the cause of the error.

On success, **munmap()** returns 0. On failure, it returns -1, and *errno* is set to indicate the cause of the error (probably to **EINVAL**).

## ERRORS

[top](#)

**EACCES** A file descriptor refers to a non-regular file. Or a file mapping was requested, but *fd* is not open for reading. Or **MAP\_SHARED** was requested and **PROT\_WRITE** is set, but *fd* is not open in read/write (**O\_RDWR**) mode. Or **PROT\_WRITE** is set, but the file is append-only.

**EAGAIN** The file has been locked, or too much memory has been locked (see [setrlimit\(2\)](#)).

**EBADF** *fd* is not a valid file descriptor (and **MAP\_ANONYMOUS** was not set).

**EINVAL** We don't like *addr*, *length*, or *offset* (e.g., they are too large, or not aligned on a page boundary).

**EINVAL** (since Linux 2.6.12) *length* was 0.

**EINVAL** *flags* contained neither **MAP\_PRIVATE** or **MAP\_SHARED**, or contained both of these values.

**ENFILE** The system-wide limit on the total number of open files has been reached.

**ENODEV** The underlying filesystem of the specified file does not support memory mapping.

**ENOMEM** No memory is available.

**ENOMEM** The process's maximum number of mappings would have been exceeded. This error can also occur for [munmap\(2\)](#), when unmapping a region in the middle of an existing mapping, since this results in two smaller mappings on either side of the region being unmapped.

#### EOVERFLOW

On 32-bit architecture together with the large file extension (i.e., using 64-bit *off\_t*): the number of pages used for *length* plus number of pages used for *offset* would overflow *unsigned long* (32 bits).

**Eperm** The *prot* argument asks for **PROT\_EXEC** but the mapped area belongs to a file on a filesystem that was mounted no-exec.

**Eperm** The operation was prevented by a file seal; see [fcntl\(2\)](#).

#### ETXTBSY

**MAP\_DENYWRITE** was set but the object specified by *fd* is open for writing.

Use of a mapped region can result in these signals:

#### SIGSEGV

Attempted write into a region mapped as read-only.

**SIGBUS** Attempted access to a portion of the buffer that does not correspond to the file (for example, beyond the end of the file, including the case where another process has truncated the file).

## ATTRIBUTES

[top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

Interface	Attribute	Value
<b>mmap()</b> , <b>munmap()</b>	Thread safety	MT-Safe

## CONFORMING TO

[top](#)

POSIX.1-2001, POSIX.1-2008, SVr4, 4.4BSD.

**AVAILABILITY**[top](#)

On POSIX systems on which `mmap()`, `msync(2)`, and `munmap()` are available, `_POSIX_MAPPED_FILES` is defined in `<unistd.h>` to a value greater than 0. (See also `sysconf(3)`.)

**NOTES**[top](#)

On some hardware architectures (e.g., i386), `PROT_WRITE` implies `PROT_READ`. It is architecture dependent whether `PROT_READ` implies `PROT_EXEC` or not. Portable programs should always set `PROT_EXEC` if they intend to execute code in the new mapping.

The portable way to create a mapping is to specify `addr` as 0 (NULL), and omit `MAP_FIXED` from `flags`. In this case, the system chooses the address for the mapping; the address is chosen so as not to conflict with any existing mapping, and will not be 0. If the `MAP_FIXED` flag is specified, and `addr` is 0 (NULL), then the mapped address will be 0 (NULL).

Certain `flags` constants are defined only if suitable feature test macros are defined (possibly by default): `_DEFAULT_SOURCE` with glibc 2.19 or later; or `_BSD_SOURCE` or `_SVID_SOURCE` in glibc 2.19 and earlier. (Requiring `_GNU_SOURCE` also suffices, and requiring that macro specifically would have been more logical, since these flags are all Linux-specific.) The relevant flags are: `MAP_32BIT`, `MAP_ANONYMOUS` (and the synonym `MAP_ANON`), `MAP_DENYWRITE`, `MAP_EXECUTABLE`, `MAP_FILE`, `MAP_GROWSDOWN`, `MAP_HUGETLB`, `MAP_LOCKED`, `MAP_NONBLOCK`, `MAP_NORESERVE`, `MAP_POPULATE`, and `MAP_STACK`.

**Timestamps changes for file-backed mappings**

For file-backed mappings, the `st_atime` field for the mapped file may be updated at any time between the `mmap()` and the corresponding unmapping; the first reference to a mapped page will update the field if it has not been already.

The `st_ctime` and `st_mtime` field for a file mapped with `PROT_WRITE` and `MAP_SHARED` will be updated after a write to the mapped region, and before a subsequent `msync(2)` with the `MS_SYNC` or `MS_ASYNC` flag, if one occurs.

**Huge page (Huge TLB) mappings**

For mappings that employ huge pages, the requirements for the arguments of `mmap()` and `munmap()` differ somewhat from the requirements for mappings that use the native system page size.

For `mmap()`, `offset` must be a multiple of the underlying huge page size. The system automatically aligns `length` to be a multiple of the underlying huge page size.

For `munmap()`, `addr` and `length` must both be a multiple of the underlying huge page size.

## C library/kernel differences

This page describes the interface provided by the glibc `mmap()` wrapper function. Originally, this function invoked a system call of the same name. Since kernel 2.4, that system call has been superseded by `mmap2(2)`, and nowadays the glibc `mmap()` wrapper function invokes `mmap2(2)` with a suitably adjusted value for `offset`.

## BUGS

[top](#)

On Linux, there are no guarantees like those suggested above under `MAP_NORESERVE`. By default, any process can be killed at any moment when the system runs out of memory.

In kernels before 2.6.7, the `MAP_POPULATE` flag has effect only if `prot` is specified as `PROT_NONE`.

SUSv3 specifies that `mmap()` should fail if `length` is 0. However, in kernels before 2.6.12, `mmap()` succeeded in this case: no mapping was created and the call returned `addr`. Since kernel 2.6.12, `mmap()` fails with the error `EINVAL` for this case.

POSIX specifies that the system shall always zero fill any partial page at the end of the object and that system will never write any modification of the object beyond its end. On Linux, when you write data to such partial page after the end of the object, the data stays in the page cache even after the file is closed and unmapped and even though the data is never written to the file itself, subsequent mappings may see the modified content. In some cases, this could be fixed by calling `msync(2)` before the unmap takes place; however, this doesn't work on tmpfs (for example, when using POSIX shared memory interface documented in `shm_overview(7)`).

## EXAMPLE

[top](#)

The following program prints part of the file specified in its first command-line argument to standard output. The range of bytes to be printed is specified via offset and length values in the second and third command-line arguments. The program creates a memory mapping of the required pages of the file and then uses `write(2)` to output the desired bytes.

### Program source

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)
```

```

int
main(int argc, char *argv[])
{
    char *addr;
    int fd;
    struct stat sb;
    off_t offset, pa_offset;
    size_t length;
    ssize_t s;

    if (argc < 3 || argc > 4) {
        fprintf(stderr, "%s file offset [length]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    fd = open(argv[1], O_RDONLY);
    if (fd == -1)
        handle_error("open");

    if (fstat(fd, &sb) == -1) /* To obtain file size */
        handle_error("fstat");

    offset = atoi(argv[2]);
    pa_offset = offset & ~(sysconf(_SC_PAGE_SIZE) - 1);
    /* offset for mmap() must be page aligned */

    if (offset >= sb.st_size) {
        fprintf(stderr, "offset is past end of file\n");
        exit(EXIT_FAILURE);
    }

    if (argc == 4) {
        length = atoi(argv[3]);
        if (offset + length > sb.st_size)
            length = sb.st_size - offset;
        /* Can't display bytes past end of file */
    } else { /* No length arg ==> display to end of file */
        length = sb.st_size - offset;
    }

    addr = mmap(NULL, length + offset - pa_offset, PROT_READ,
               MAP_PRIVATE, fd, pa_offset);
    if (addr == MAP_FAILED)
        handle_error("mmap");

    s = write(STDOUT_FILENO, addr + offset - pa_offset, length);
    if (s != length) {
        if (s == -1)
            handle_error("write");

        fprintf(stderr, "partial write");
        exit(EXIT_FAILURE);
    }
}

```

```

    }

    munmap(addr, length + offset - pa_offset);
    close(fd);

    exit(EXIT_SUCCESS);
}

```

**SEE ALSO**[top](#)

[getpagesize\(2\)](#), [memfd\\_create\(2\)](#), [mincore\(2\)](#), [mlock\(2\)](#), [mmap2\(2\)](#), [mprotect\(2\)](#), [mremap\(2\)](#), [msync\(2\)](#), [remap\\_file\\_pages\(2\)](#), [setrlimit\(2\)](#), [shmat\(2\)](#), [shm\\_open\(3\)](#), [shm\\_overview\(7\)](#)

The descriptions of the following files in [proc\(5\)](#): [/proc/\[pid\]/maps](#), [/proc/\[pid\]/map\\_files](#), and [/proc/\[pid\]/smaps](#).

B.O. Gallmeister, POSIX.4, O'Reilly, pp. 128–129 and 389–391.

**COLOPHON**[top](#)

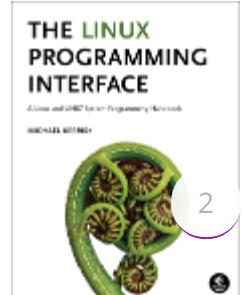
This page is part of release 4.08 of the Linux [man-pages](#) project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at <https://www.kernel.org/doc/man-pages/>.

**Linux****2016-07-17****MMAP(2)**[Copyright and license for this manual page](#)

HTML rendering created 2016-10-08 by Michael Kerrisk, author of [The Linux Programming Interface](#), maintainer of the [Linux man-pages](#) project.

For details of in-depth Linux/UNIX system programming training courses that I teach, look [here](#).

Hosting by [jambit GmbH](#).



## **Part II**

# **Concurrency**



## A Dialogue on Concurrency

**Professor:** *And thus we reach the second of our three pillars of operating systems: concurrency.*

**Student:** *I thought there were four pillars...?*

**Professor:** *Nope, that was in an older version of the book.*

**Student:** *Umm... OK. So what is concurrency, oh wonderful professor?*

**Professor:** *Well, imagine we have a peach —*

**Student:** *(interrupting) Peaches again! What is it with you and peaches?*

**Professor:** *Ever read T.S. Eliot? The Love Song of J. Alfred Prufrock, “Do I dare to eat a peach”, and all that fun stuff?*

**Student:** *Oh yes! In English class in high school. Great stuff! I really liked the part where —*

**Professor:** *(interrupting) This has nothing to do with that — I just like peaches. Anyhow, imagine there are a lot of peaches on a table, and a lot of people who wish to eat them. Let’s say we did it this way: each eater first identifies a peach visually, and then tries to grab it and eat it. What is wrong with this approach?*

**Student:** *Hmmm... seems like you might see a peach that somebody else also sees. If they get there first, when you reach out, no peach for you!*

**Professor:** *Exactly! So what should we do about it?*

**Student:** *Well, probably develop a better way of going about this. Maybe form a line, and when you get to the front, grab a peach and get on with it.*

**Professor:** *Good! But what’s wrong with your approach?*

**Student:** *Sheesh, do I have to do all the work?*

**Professor:** *Yes.*

**Student:** *OK, let me think. Well, we used to have many people grabbing for peaches all at once, which is faster. But in my way, we just go one at a time, which is correct, but quite a bit slower. The best kind of approach would be fast and correct, probably.*

**Professor:** You are really starting to impress. In fact, you just told us everything we need to know about concurrency! Well done.

**Student:** I did? I thought we were just talking about peaches. Remember, this is usually a part where you make it about computers again.

**Professor:** Indeed. My apologies! One must never forget the concrete. Well, as it turns out, there are certain types of programs that we call **multi-threaded** applications; each **thread** is kind of like an independent agent running around in this program, doing things on the program's behalf. But these threads access memory, and for them, each spot of memory is kind of like one of those peaches. If we don't coordinate access to memory between threads, the program won't work as expected. Make sense?

**Student:** Kind of. But why do we talk about this in an OS class? Isn't that just application programming?

**Professor:** Good question! A few reasons, actually. First, the OS must support multi-threaded applications with primitives such as **locks** and **condition variables**, which we'll talk about soon. Second, the OS itself was the first concurrent program — it must access its own memory very carefully or many strange and terrible things will happen. Really, it can get quite grisly.

**Student:** I see. Sounds interesting. There are more details, I imagine?

**Professor:** Indeed there are...

## Concurrency: An Introduction

Thus far, we have seen the development of the basic abstractions that the OS performs. We have seen how to take a single physical CPU and turn it into multiple **virtual CPUs**, thus enabling the illusion of multiple programs running at the same time. We have also seen how to create the illusion of a large, private **virtual memory** for each process; this abstraction of the **address space** enables each program to behave as if it has its own memory when indeed the OS is secretly multiplexing address spaces across physical memory (and sometimes, disk).

In this note, we introduce a new abstraction for a single running process: that of a **thread**. Instead of our classic view of a single point of execution within a program (i.e., a single PC where instructions are being fetched from and executed), a **multi-threaded** program has more than one point of execution (i.e., multiple PCs, each of which is being fetched and executed from). Perhaps another way to think of this is that each thread is very much like a separate process, except for one difference: they *share* the same address space and thus can access the same data.

The state of a single thread is thus very similar to that of a process. It has a program counter (PC) that tracks where the program is fetching instructions from. Each thread has its own private set of registers it uses for computation; thus, if there are two threads that are running on a single processor, when switching from running one (T1) to running the other (T2), a **context switch** must take place. The context switch between threads is quite similar to the context switch between processes, as the register state of T1 must be saved and the register state of T2 restored before running T2. With processes, we saved state to a **process control block (PCB)**; now, we'll need one or more **thread control blocks (TCBs)** to store the state of each thread of a process. There is one major difference, though, in the context switch we perform between threads as compared to processes: the address space remains the same (i.e., there is no need to switch which page table we are using).

One other major difference between threads and processes concerns the stack. In our simple model of the address space of a classic process (which we can now call a **single-threaded** process), there is a single stack, usually residing at the bottom of the address space (Figure 26.1, left).

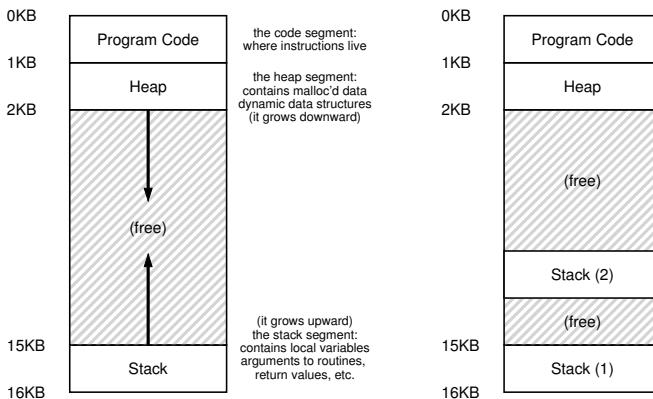


Figure 26.1: Single-Threaded And Multi-Threaded Address Spaces

However, in a multi-threaded process, each thread runs independently and of course may call into various routines to do whatever work it is doing. Instead of a single stack in the address space, there will be one per thread. Let's say we have a multi-threaded process that has two threads in it; the resulting address space looks different (Figure 26.1, right).

In this figure, you can see two stacks spread throughout the address space of the process. Thus, any stack-allocated variables, parameters, return values, and other things that we put on the stack will be placed in what is sometimes called **thread-local** storage, i.e., the stack of the relevant thread.

You might also notice how this ruins our beautiful address space layout. Before, the stack and heap could grow independently and trouble only arose when you ran out of room in the address space. Here, we no longer have such a nice situation. Fortunately, this is usually OK, as stacks do not generally have to be very large (the exception being in programs that make heavy use of recursion).

## 26.1 Why Use Threads?

Before getting into the details of threads and some of the problems you might have in writing multi-threaded programs, let's first answer a more simple question. Why should you use threads at all?

As it turns out, there are at least two major reasons you should use threads. The first is simple: **parallelism**. Imagine you are writing a program that performs operations on very large arrays, for example, adding two large arrays together, or incrementing the value of each element in the array by some amount. If you are running on just a single processor, the task is straightforward: just perform each operation and be done. However, if you are executing the program on a system with multiple

processors, you have the potential of speeding up this process considerably by using the processors to each perform a portion of the work. The task of transforming your standard **single-threaded** program into a program that does this sort of work on multiple CPUs is called **parallelization**, and using a thread per CPU to do this work is a natural and typical way to make programs run faster on modern hardware.

The second reason is a bit more subtle: to avoid blocking program progress due to slow I/O. Imagine that you are writing a program that performs different types of I/O: either waiting to send or receive a message, for an explicit disk I/O to complete, or even (implicitly) for a page fault to finish. Instead of waiting, your program may wish to do something else, including utilizing the CPU to perform computation, or even issuing further I/O requests. Using threads is a natural way to avoid getting stuck; while one thread in your program waits (i.e., is blocked waiting for I/O), the CPU scheduler can switch to other threads, which are ready to run and do something useful. Threading enables **overlap** of I/O with other activities *within* a single program, much like **multiprogramming** did for processes *across* programs; as a result, many modern server-based applications (web servers, database management systems, and the like) make use of threads in their implementations.

Of course, in either of the cases mentioned above, you could use multiple *processes* instead of threads. However, threads share an address space and thus make it easy to share data, and hence are a natural choice when constructing these types of programs. Processes are a more sound choice for logically separate tasks where little sharing of data structures in memory is needed.

## 26.2 An Example: Thread Creation

Let's get into some of the details. Say we wanted to run a program that creates two threads, each of which does some independent work, in this case printing "A" or "B". The code is shown in Figure 26.2 (page 4).

The main program creates two threads, each of which will run the function `mythread()`, though with different arguments (the string A or B). Once a thread is created, it may start running right away (depending on the whims of the scheduler); alternately, it may be put in a "ready" but not "running" state and thus not run yet. Of course, on a multiprocessor, the threads could even be running at the same time, but let's not worry about this possibility quite yet.

After creating the two threads (let's call them T1 and T2), the main thread calls `pthread.join()`, which waits for a particular thread to complete. It does so twice, thus ensuring T1 and T2 will run and complete before finally allowing the main thread to run again; when it does, it will print "main: end" and exit. Overall, three threads were employed during this run: the main thread, T1, and T2.

```

1 #include <stdio.h>
2 #include <assert.h>
3 #include <pthread.h>
4
5 void *mythread(void *arg) {
6     printf("%s\n", (char *) arg);
7     return NULL;
8 }
9
10 int
11 main(int argc, char *argv[]) {
12     pthread_t p1, p2;
13     int rc;
14     printf("main: begin\n");
15     rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
16     rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
17     // join waits for the threads to finish
18     rc = pthread_join(p1, NULL); assert(rc == 0);
19     rc = pthread_join(p2, NULL); assert(rc == 0);
20     printf("main: end\n");
21     return 0;
22 }
```

Figure 26.2: Simple Thread Creation Code (t0.c)

Let us examine the possible execution ordering of this little program. In the execution diagram (Figure 26.3, page 5), time increases in the downwards direction, and each column shows when a different thread (the main one, or Thread 1, or Thread 2) is running.

Note, however, that this ordering is not the only possible ordering. In fact, given a sequence of instructions, there are quite a few, depending on which thread the scheduler decides to run at a given point. For example, once a thread is created, it may run immediately, which would lead to the execution shown in Figure 26.4 (page 5).

We also could even see “B” printed before “A”, if, say, the scheduler decided to run Thread 2 first even though Thread 1 was created earlier; there is no reason to assume that a thread that is created first will run first. Figure 26.5 (page 5) shows this final execution ordering, with Thread 2 getting to strut its stuff before Thread 1.

As you might be able to see, one way to think about thread creation is that it is a bit like making a function call; however, instead of first executing the function and then returning to the caller, the system instead creates a new thread of execution for the routine that is being called, and it runs independently of the caller, perhaps before returning from the create, but perhaps much later. What runs next is determined by the OS **scheduler**, and although the scheduler likely implements some sensible algorithm, it is hard to know what will run at any given moment in time.

As you also might be able to tell from this example, threads make life complicated: it is already hard to tell what will run when! Computers are hard enough to understand without concurrency. Unfortunately, with concurrency, it simply gets worse. Much worse.

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
	runs	
	prints "B"	
	returns	
prints "main: end"		

Figure 26.3: Thread Trace (1)

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
	runs	
	prints "B"	
	returns	
waits for T1		
<i>returns immediately; T1 is done</i>		
waits for T2		
<i>returns immediately; T2 is done</i>		
prints "main: end"		

Figure 26.4: Thread Trace (2)

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
	runs	
	prints "B"	
	returns	
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
<i>returns immediately; T2 is done</i>		
prints "main: end"		

Figure 26.5: Thread Trace (3)

```

1 #include <stdio.h>
2 #include <pthread.h>
3 #include "mythreads.h"
4
5 static volatile int counter = 0;
6
7 //
8 // mythread()
9 //
10 // Simply adds 1 to counter repeatedly, in a loop
11 // No, this is not how you would add 10,000,000 to
12 // a counter, but it shows the problem nicely.
13 //
14 void *
15 mythread(void *arg)
16 {
17     printf("%s: begin\n", (char *) arg);
18     int i;
19     for (i = 0; i < 1e7; i++) {
20         counter = counter + 1;
21     }
22     printf("%s: done\n", (char *) arg);
23     return NULL;
24 }
25
26 //
27 // main()
28 //
29 // Just launches two threads (pthread_create)
30 // and then waits for them (pthread_join)
31 //
32 int
33 main(int argc, char *argv[])
34 {
35     pthread_t p1, p2;
36     printf("main: begin (counter = %d)\n", counter);
37     Pthread_create(&p1, NULL, mythread, "A");
38     Pthread_create(&p2, NULL, mythread, "B");
39
40     // join waits for the threads to finish
41     Pthread_join(p1, NULL);
42     Pthread_join(p2, NULL);
43     printf("main: done with both (counter = %d)\n", counter);
44     return 0;
45 }

```

Figure 26.6: Sharing Data: Uh Oh (t1.c)

### 26.3 Why It Gets Worse: Shared Data

The simple thread example we showed above was useful in showing how threads are created and how they can run in different orders depending on how the scheduler decides to run them. What it doesn't show you, though, is how threads interact when they access shared data.

Let us imagine a simple example where two threads wish to update a global shared variable. The code we'll study is in Figure 26.6 (page 6).

Here are a few notes about the code. First, as Stevens suggests [SR05], we wrap the thread creation and join routines to simply exit on failure; for a program as simple as this one, we want to at least notice an error occurred (if it did), but not do anything very smart about it (e.g., just exit). Thus, `Pthread_create()` simply calls `pthread_create()` and makes sure the return code is 0; if it isn't, `Pthread_create()` just prints a message and exits.

Second, instead of using two separate function bodies for the worker threads, we just use a single piece of code, and pass the thread an argument (in this case, a string) so we can have each thread print a different letter before its messages.

Finally, and most importantly, we can now look at what each worker is trying to do: add a number to the shared variable `counter`, and do so 10 million times ( $1e7$ ) in a loop. Thus, the desired final result is: 20,000,000.

We now compile and run the program, to see how it behaves. Sometimes, everything works how we might expect:

```
prompt> gcc -o main main.c -Wall -pthread
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

Unfortunately, when we run this code, even on a single processor, we don't necessarily get the desired result. Sometimes, we get:

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

Let's try it one more time, just to see if we've gone crazy. After all, aren't computers supposed to produce **deterministic** results, as you have been taught?! Perhaps your professors have been lying to you? (*gasp*)

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

Not only is each run wrong, but also yields a *different* result! A big question remains: why does this happen?

**TIP: KNOW AND USE YOUR TOOLS**

You should always learn new tools that help you write, debug, and understand computer systems. Here, we use a neat tool called a **disassembler**. When you run a disassembler on an executable, it shows you what assembly instructions make up the program. For example, if we wish to understand the low-level code to update a counter (as in our example), we run `objdump` (Linux) to see the assembly code:

```
prompt> objdump -d main
```

Doing so produces a long listing of all the instructions in the program, neatly labeled (particularly if you compiled with the `-g` flag), which includes symbol information in the program. The `objdump` program is just one of many tools you should learn how to use; a debugger like `gdb`, memory profilers like `valgrind` or `purify`, and of course the compiler itself are others that you should spend time to learn more about; the better you are at using your tools, the better systems you'll be able to build.

## 26.4 The Heart Of The Problem: Uncontrolled Scheduling

To understand why this happens, we must understand the code sequence that the compiler generates for the update to `counter`. In this case, we wish to simply add a number (1) to `counter`. Thus, the code sequence for doing so might look something like this (in x86);

```
mov 0x8049a1c, %eax  
add $0x1, %eax  
mov %eax, 0x8049a1c
```

This example assumes that the variable `counter` is located at address `0x8049a1c`. In this three-instruction sequence, the x86 `mov` instruction is used first to get the memory value at the address and put it into register `eax`. Then, the `add` is performed, adding 1 (`0x1`) to the contents of the `eax` register, and finally, the contents of `eax` are stored back into memory at the same address.

Let us imagine one of our two threads (Thread 1) enters this region of code, and is thus about to increment `counter` by one. It loads the value of `counter` (let's say it's 50 to begin with) into its register `eax`. Thus, `eax=50` for Thread 1. Then it adds one to the register; thus `eax=51`. Now, something unfortunate happens: a timer interrupt goes off; thus, the OS saves the state of the currently running thread (its PC, its registers including `eax`, etc.) to the thread's TCB.

Now something worse happens: Thread 2 is chosen to run, and it enters this same piece of code. It also executes the first instruction, getting the value of `counter` and putting it into its `eax` (remember: each thread when running has its own private registers; the registers are **virtualized** by the context-switch code that saves and restores them). The value of

OS	Thread 1	Thread 2	(after instruction)		
			PC	%eax	counter
		<i>before critical section</i>	100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add \$0x1, %eax	108	51	50
<b>interrupt</b>	<i>save T1's state</i>				
	<i>restore T2's state</i>		100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add \$0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
<b>interrupt</b>	<i>save T2's state</i>				
	<i>restore T1's state</i>		108	51	51
		mov %eax, 0x8049a1c	113	51	51

Figure 26.7: The Problem: Up Close and Personal

counter is still 50 at this point, and thus Thread 2 has eax=50. Let's then assume that Thread 2 executes the next two instructions, incrementing eax by 1 (thus eax=51), and then saving the contents of eax into counter (address 0x8049a1c). Thus, the global variable counter now has the value 51.

Finally, another context switch occurs, and Thread 1 resumes running. Recall that it had just executed the mov and add, and is now about to perform the final mov instruction. Recall also that eax=51. Thus, the final mov instruction executes, and saves the value to memory; the counter is set to 51 again.

Put simply, what has happened is this: the code to increment counter has been run twice, but counter, which started at 50, is now only equal to 51. A “correct” version of this program should have resulted in the variable counter equal to 52.

Let's look at a detailed execution trace to understand the problem better. Assume, for this example, that the above code is loaded at address 100 in memory, like the following sequence (note for those of you used to nice, RISC-like instruction sets: x86 has variable-length instructions; this mov instruction takes up 5 bytes of memory, and the add only 3):

```
100 mov    0x8049a1c, %eax
105 add    $0x1, %eax
108 mov    %eax, 0x8049a1c
```

With these assumptions, what happens is shown in Figure 26.7. Assume the counter starts at value 50, and trace through this example to make sure you understand what is going on.

What we have demonstrated here is called a **race condition**: the results depend on the timing execution of the code. With some bad luck (i.e., context switches that occur at untimely points in the execution), we get the wrong result. In fact, we may get a different result each time; thus, instead of a nice **deterministic** computation (which we are used to from computers), we call this result **indeterminate**, where it is not known what the output will be and it is indeed likely to be different across runs.

Because multiple threads executing this code can result in a race condition, we call this code a **critical section**. A critical section is a piece of code that accesses a shared variable (or more generally, a shared resource) and must not be concurrently executed by more than one thread.

What we really want for this code is what we call **mutual exclusion**. This property guarantees that if one thread is executing within the critical section, the others will be prevented from doing so.

Virtually all of these terms, by the way, were coined by Edsger Dijkstra, who was a pioneer in the field and indeed won the Turing Award because of this and other work; see his 1968 paper on “Cooperating Sequential Processes” [D68] for an amazingly clear description of the problem. We’ll be hearing more about Dijkstra in this section of the book.

## 26.5 The Wish For Atomicity

One way to solve this problem would be to have more powerful instructions that, in a single step, did exactly whatever we needed done and thus removed the possibility of an untimely interrupt. For example, what if we had a super instruction that looked like this?

```
memory-add 0x8049a1c, $0x1
```

Assume this instruction adds a value to a memory location, and the hardware guarantees that it executes **atomically**; when the instruction executed, it would perform the update as desired. It could not be interrupted mid-instruction, because that is precisely the guarantee we receive from the hardware: when an interrupt occurs, either the instruction has not run at all, or it has run to completion; there is no in-between state. Hardware can be a beautiful thing, no?

Atomically, in this context, means “as a unit”, which sometimes we take as “all or none.” What we’d like is to execute the three instruction sequence atomically:

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

As we said, if we had a single instruction to do this, we could just issue that instruction and be done. But in the general case, we won’t have such an instruction. Imagine we were building a concurrent B-tree, and wished to update it; would we really want the hardware to support an “atomic update of B-tree” instruction? Probably not, at least in a sane instruction set.

Thus, what we will instead do is ask the hardware for a few useful instructions upon which we can build a general set of what we call **synchronization primitives**. By using these hardware synchronization primitives, in combination with some help from the operating system, we will be able to build multi-threaded code that accesses critical sections in a

**TIP: USE ATOMIC OPERATIONS**

Atomic operations are one of the most powerful underlying techniques in building computer systems, from the computer architecture, to concurrent code (what we are studying here), to file systems (which we'll study soon enough), database management systems, and even distributed systems [L+93].

The idea behind making a series of actions **atomic** is simply expressed with the phrase “all or nothing”; it should either appear as if all of the actions you wish to group together occurred, or that none of them occurred, with no in-between state visible. Sometimes, the grouping of many actions into a single atomic action is called a **transaction**, an idea developed in great detail in the world of databases and transaction processing [GR92].

In our theme of exploring concurrency, we'll be using synchronization primitives to turn short sequences of instructions into atomic blocks of execution, but the idea of atomicity is much bigger than that, as we will see. For example, file systems use techniques such as journaling or copy-on-write in order to atomically transition their on-disk state, critical for operating correctly in the face of system failures. If that doesn't make sense, don't worry — it will, in some future chapter.

synchronized and controlled manner, and thus reliably produces the correct result despite the challenging nature of concurrent execution. Pretty awesome, right?

This is the problem we will study in this section of the book. It is a wonderful and hard problem, and should make your mind hurt (a bit). If it doesn't, then you don't understand! Keep working until your head hurts; you then know you're headed in the right direction. At that point, take a break; we don't want your head hurting too much.

**THE CRUX:**  
**HOW TO PROVIDE SUPPORT FOR SYNCHRONIZATION**

What support do we need from the hardware in order to build useful synchronization primitives? What support do we need from the OS? How can we build these primitives correctly and efficiently? How can programs use them to get the desired results?

## 26.6 One More Problem: Waiting For Another

This chapter has set up the problem of concurrency as if only one type of interaction occurs between threads, that of accessing shared variables and the need to support atomicity for critical sections. As it turns out, there is another common interaction that arises, where one thread must wait for another to complete some action before it continues. This interaction arises, for example, when a process performs a disk I/O and is put to sleep; when the I/O completes, the process needs to be roused from its slumber so it can continue.

Thus, in the coming chapters, we'll be not only studying how to build support for synchronization primitives to support atomicity but also for mechanisms to support this type of sleeping/waking interaction that is common in multi-threaded programs. If this doesn't make sense right now, that is OK! It will soon enough, when you read the chapter on **condition variables**. If it doesn't by then, well, then it is less OK, and you should read that chapter again (and again) until it does make sense.

## 26.7 Summary: Why in OS Class?

Before wrapping up, one question that you might have is: why are we studying this in OS class? "History" is the one-word answer; the OS was the first concurrent program, and many techniques were created for use *within* the OS. Later, with multi-threaded processes, application programmers also had to consider such things.

For example, imagine the case where there are two processes running. Assume they both call `write()` to write to the file, and both wish to append the data to the file (i.e., add the data to the end of the file, thus increasing its length). To do so, both must allocate a new block, record in the inode of the file where this block lives, and change the size of the file to reflect the new larger size (among other things; we'll learn more about files in the third part of the book). Because an interrupt may occur at any time, the code that updates these shared structures (e.g., a bitmap for allocation, or the file's inode) are critical sections; thus, OS designers, from the very beginning of the introduction of the interrupt, had to worry about how the OS updates internal structures. An untimely interrupt causes all of the problems described above. Not surprisingly, page tables, process lists, file system structures, and virtually every kernel data structure has to be carefully accessed, with the proper synchronization primitives, to work correctly.

ASIDE: KEY CONCURRENCY TERMS  
CRITICAL SECTION, RACE CONDITION,  
INDETERMINATE, MUTUAL EXCLUSION

These four terms are so central to concurrent code that we thought it worth while to call them out explicitly. See some of Dijkstra's early work [D65,D68] for more details.

- A **critical section** is a piece of code that accesses a *shared* resource, usually a variable or data structure.
- A **race condition** arises if multiple threads of execution enter the critical section at roughly the same time; both attempt to update the shared data structure, leading to a surprising (and perhaps undesirable) outcome.
- An **ineterminate** program consists of one or more race conditions; the output of the program varies from run to run, depending on which threads ran when. The outcome is thus not **deterministic**, something we usually expect from computer systems.
- To avoid these problems, threads should use some kind of **mutual exclusion** primitives; doing so guarantees that only a single thread ever enters a critical section, thus avoiding races, and resulting in deterministic program outputs.

## References

- [D65] "Solution of a problem in concurrent programming control"  
 E. W. Dijkstra  
*Communications of the ACM*, 8(9):569, September 1965  
*Pointed to as the first paper of Dijkstra's where he outlines the mutual exclusion problem and a solution. The solution, however, is not widely used; advanced hardware and OS support is needed, as we will see in the coming chapters.*
- [D68] "Cooperating sequential processes"  
 Edsger W. Dijkstra, 1968  
 Available: <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF>  
*Dijkstra has an amazing number of his old papers, notes, and thoughts recorded (for posterity) on this website at the last place he worked, the University of Texas. Much of his foundational work, however, was done years earlier while he was at the Technische Hochschule of Eindhoven (THE), including this famous paper on "cooperating sequential processes", which basically outlines all of the thinking that has to go into writing multi-threaded programs. Dijkstra discovered much of this while working on an operating system named after his school: the "THE" operating system (said "T", "H", "E", and not like the word "the").*
- [GR92] "Transaction Processing: Concepts and Techniques"  
 Jim Gray and Andreas Reuter  
 Morgan Kaufmann, September 1992  
*This book is the bible of transaction processing, written by one of the legends of the field, Jim Gray. It is, for this reason, also considered Jim Gray's "brain dump", in which he wrote down everything he knows about how database management systems work. Sadly, Gray passed away tragically a few years back, and many of us lost a friend and great mentor, including the co-authors of said book, who were lucky enough to interact with Gray during their graduate school years.*
- [L+93] "Atomic Transactions"  
 Nancy Lynch, Michael Merritt, William Weihl, Alan Fekete  
 Morgan Kaufmann, August 1993  
*A nice text on some of the theory and practice of atomic transactions for distributed systems. Perhaps a bit formal for some, but lots of good material is found herein.*
- [SR05] "Advanced Programming in the UNIX Environment"  
 W. Richard Stevens and Stephen A. Rago  
 Addison-Wesley, 2005  
*As we've said many times, buy this book, and read it, in little chunks, preferably before going to bed. This way, you will actually fall asleep more quickly; more importantly, you learn a little more about how to become a serious UNIX programmer.*

## Homework

This program, `x86.py`, allows you to see how different thread interleavings either cause or avoid race conditions. See the README for details on how the program works and its basic inputs, then answer the questions below.

### Questions

1. To start, let's examine a simple program, "loop.s". First, just look at the program, and see if you can understand it: `cat loop.s`. Then, run it with these arguments:

```
./x86.py -p loop.s -t 1 -i 100 -R dx
```

This specifies a single thread, an interrupt every 100 instructions, and tracing of register `%dx`. Can you figure out what the value of `%dx` will be during the run? Once you have, run the same above and use the `-c` flag to check your answers; note the answers, on the left, show the value of the register (or memory value) *after* the instruction on the right has run.

2. Now run the same code but with these flags:

```
./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx
```

This specifies two threads, and initializes each `%dx` register to 3. What values will `%dx` see? Run with the `-c` flag to see the answers. Does the presence of multiple threads affect anything about your calculations? Is there a race condition in this code?

3. Now run the following:

```
./x86.py -p loop.s -t 2 -i 3 -r -a dx=3,dx=3 -R dx
```

This makes the interrupt interval quite small and random; use different seeds with `-s` to see different interleavings. Does the frequency of interruption change anything about this program?

4. Next we'll examine a different program (`looping-race-nolock.s`). This program accesses a shared variable located at memory address 2000; we'll call this variable `x` for simplicity. Run it with a single thread and make sure you understand what it does, like this:

```
./x86.py -p looping-race-nolock.s -t 1 -M 2000
```

What value is found in `x` (i.e., at memory address 2000) throughout the run? Use `-c` to check your answer.

5. Now run with multiple iterations and threads:

```
./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000
```

Do you understand why the code in each thread loops three times? What will the final value of `x` be?

6. Now run with random interrupt intervals:

```
./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0
```

Then change the random seed, setting `-s 1`, then `-s 2`, etc. Can you tell, just by looking at the thread interleaving, what the final value of `x` will be? Does the exact location of the interrupt matter? Where can it safely occur? Where does an interrupt cause trouble? In other words, where is the critical section exactly?

7. Now use a fixed interrupt interval to explore the program further. Run:

```
./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1
```

See if you can guess what the final value of the shared variable `x` will be. What about when you change `-i 2`, `-i 3`, etc.? For which interrupt intervals does the program give the “correct” final answer?

8. Now run the same code for more loops (e.g., set `-a bx=100`). What interrupt intervals, set with the `-i` flag, lead to a “correct” outcome? Which intervals lead to surprising results?
9. We’ll examine one last program in this homework (`wait-for-me.s`). Run the code like this:

```
./x86.py -p wait-for-me.s -a ax=1,ax=0 -R ax -M 2000
```

This sets the `%ax` register to 1 for thread 0, and 0 for thread 1, and watches the value of `%ax` and memory location 2000 throughout the run. How should the code behave? How is the value at location 2000 being used by the threads? What will its final value be?

10. Now switch the inputs:

```
./x86.py -p wait-for-me.s -a ax=0,ax=1 -R ax -M 2000
```

How do the threads behave? What is thread 0 doing? How would changing the interrupt interval (e.g., `-i 1000`, or perhaps to use random intervals) change the trace outcome? Is the program efficiently using the CPU?

## Interlude: Thread API

This chapter briefly covers the main portions of the thread API. Each part will be explained further in the subsequent chapters, as we show how to use the API. More details can be found in various books and online sources [B89, B97, B+96, K+96]. We should note that the subsequent chapters introduce the concepts of locks and condition variables more slowly, with many examples; this chapter is thus better used as a reference.

### CRUX: HOW TO CREATE AND CONTROL THREADS

What interfaces should the OS present for thread creation and control? How should these interfaces be designed to enable ease of use as well as utility?

### 27.1 Thread Creation

The first thing you have to be able to do to write a multi-threaded program is to create new threads, and thus some kind of thread creation interface must exist. In POSIX, it is easy:

```
#include <pthread.h>
int
pthread_create(      pthread_t *      thread,
                    const pthread_attr_t * attr,
                    void *                  (*start_routine)(void*),
                    void *                  arg);
```

This declaration might look a little complex (particularly if you haven't used function pointers in C), but actually it's not too bad. There are four arguments: `thread`, `attr`, `start_routine`, and `arg`. The first, `thread`, is a pointer to a structure of type `pthread_t`; we'll use this structure to interact with this thread, and thus we need to pass it to `pthread_create()` in order to initialize it.

The second argument, `attr`, is used to specify any attributes this thread might have. Some examples include setting the stack size or perhaps information about the scheduling priority of the thread. An attribute is initialized with a separate call to `pthread_attr_init()`; see the manual page for details. However, in most cases, the defaults will be fine; in this case, we will simply pass the value `NULL` in.

The third argument is the most complex, but is really just asking: which function should this thread start running in? In C, we call this a **function pointer**, and this one tells us the following is expected: a function name (`start_routine`), which is passed a single argument of type `void *` (as indicated in the parentheses after `start_routine`), and which returns a value of type `void *` (i.e., a **void pointer**).

If this routine instead required an integer argument, instead of a void pointer, the declaration would look like this:

```
int pthread_create(..., // first two args are the same
                  void *      (*start_routine)(int),
                  int         arg);
```

If instead the routine took a void pointer as an argument, but returned an integer, it would look like this:

```
int pthread_create(..., // first two args are the same
                  int       (*start_routine)(void *),
                  void *    arg);
```

Finally, the fourth argument, `arg`, is exactly the argument to be passed to the function where the thread begins execution. You might ask: why do we need these void pointers? Well, the answer is quite simple: having a void pointer as an argument to the function `start_routine` allows us to pass in *any* type of argument; having it as a return value allows the thread to return *any* type of result.

Let's look at an example in Figure 27.1. Here we just create a thread that is passed two arguments, packaged into a single type we define ourselves (`myarg_t`). The thread, once created, can simply cast its argument to the type it expects and thus unpack the arguments as desired.

And there it is! Once you create a thread, you really have another live executing entity, complete with its own call stack, running within the *same* address space as all the currently existing threads in the program. The fun thus begins!

## 27.2 Thread Completion

The example above shows how to create a thread. However, what happens if you want to wait for a thread to complete? You need to do something special in order to wait for completion; in particular, you must call the routine `pthread_join()`.

```
int pthread_join(pthread_t thread, void **value_ptr);
```

```

1 #include <pthread.h>
2
3 typedef struct __myarg_t {
4     int a;
5     int b;
6 } myarg_t;
7
8 void *mythread(void *arg) {
9     myarg_t *m = (myarg_t *) arg;
10    printf("%d %d\n", m->a, m->b);
11    return NULL;
12 }
13
14 int
15 main(int argc, char *argv[]) {
16     pthread_t p;
17     int rc;
18
19     myarg_t args;
20     args.a = 10;
21     args.b = 20;
22     rc = pthread_create(&p, NULL, mythread, &args);
23     ...
24 }
```

Figure 27.1: Creating a Thread

This routine takes two arguments. The first is of type `pthread_t`, and is used to specify which thread to wait for. This variable is initialized by the thread creation routine (when you pass a pointer to it as an argument to `pthread_create()`); if you keep it around, you can use it to wait for that thread to terminate.

The second argument is a pointer to the return value you expect to get back. Because the routine can return anything, it is defined to return a pointer to void; because the `pthread_join()` routine *changes* the value of the passed in argument, you need to pass in a pointer to that value, not just the value itself.

Let's look at another example (Figure 27.2). In the code, a single thread is again created, and passed a couple of arguments via the `myarg_t` structure. To return values, the `myret_t` type is used. Once the thread is finished running, the main thread, which has been waiting inside of the `pthread_join()` routine<sup>1</sup>, then returns, and we can access the values returned from the thread, namely whatever is in `myret_t`.

A few things to note about this example. First, often times we don't have to do all of this painful packing and unpacking of arguments. For example, if we just create a thread with no arguments, we can pass `NULL` in as an argument when the thread is created. Similarly, we can pass `NULL` into `pthread_join()` if we don't care about the return value.

Second, if we are just passing in a single value (e.g., an `int`), we don't

---

<sup>1</sup>Note we use wrapper functions here; specifically, we call `Malloc()`, `Pthread.join()`, and `Pthread.create()`, which just call their similarly-named lower-case versions and make sure the routines did not return anything unexpected.

```

1  #include <stdio.h>
2  #include <pthread.h>
3  #include <assert.h>
4  #include <stdlib.h>
5
6  typedef struct __myarg_t {
7      int a;
8      int b;
9  } myarg_t;
10
11 typedef struct __myret_t {
12     int x;
13     int y;
14 } myret_t;
15
16 void *mythread(void *arg) {
17     myarg_t *m = (myarg_t *) arg;
18     printf("%d %d\n", m->a, m->b);
19     myret_t *r = Malloc(sizeof(myret_t));
20     r->x = 1;
21     r->y = 2;
22     return (void *) r;
23 }
24
25 int
26 main(int argc, char *argv[]) {
27     int rc;
28     pthread_t p;
29     myret_t *m;
30
31     myarg_t args;
32     args.a = 10;
33     args.b = 20;
34     Pthread_create(&p, NULL, mythread, &args);
35     Pthread_join(p, (void **) &m);
36     printf("returned %d %d\n", m->x, m->y);
37     return 0;
38 }
```

Figure 27.2: Waiting for Thread Completion

have to package it up as an argument. Figure 27.3 shows an example. In this case, life is a bit simpler, as we don't have to package arguments and return values inside of structures.

Third, we should note that one has to be extremely careful with how values are returned from a thread. In particular, never return a pointer which refers to something allocated on the thread's call stack. If you do, what do you think will happen? (think about it!) Here is an example of a dangerous piece of code, modified from the example in Figure 27.2.

```

1 void *mythread(void *arg) {
2     myarg_t *m = (myarg_t *) arg;
3     printf("%d %d\n", m->a, m->b);
4     myret_t r; // ALLOCATED ON STACK: BAD!
5     r.x = 1;
6     r.y = 2;
7     return (void *) &r;
8 }
```

```

void *mythread(void *arg) {
    int m = (int) arg;
    printf("%d\n", m);
    return (void *) (arg + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc, m;
    Pthread_create(&p, NULL, mythread, (void *) 100);
    Pthread_join(p, (void **) &m);
    printf("returned %d\n", m);
    return 0;
}

```

Figure 27.3: Simpler Argument Passing to a Thread

In this case, the variable `r` is allocated on the stack of `mythread`. However, when it returns, the value is automatically deallocated (that's why the stack is so easy to use, after all!), and thus, passing back a pointer to a now deallocated variable will lead to all sorts of bad results. Certainly, when you print out the values you think you returned, you'll probably (but not necessarily!) be surprised. Try it and find out for yourself<sup>2</sup>!

Finally, you might notice that the use of `pthread_create()` to create a thread, followed by an immediate call to `pthread_join()`, is a pretty strange way to create a thread. In fact, there is an easier way to accomplish this exact task; it's called a **procedure call**. Clearly, we'll usually be creating more than just one thread and waiting for it to complete, otherwise there is not much purpose to using threads at all.

We should note that not all code that is multi-threaded uses the join routine. For example, a multi-threaded web server might create a number of worker threads, and then use the main thread to accept requests and pass them to the workers, indefinitely. Such long-lived programs thus may not need to join. However, a parallel program that creates threads to execute a particular task (in parallel) will likely use join to make sure all such work completes before exiting or moving onto the next stage of computation.

## 27.3 Locks

Beyond thread creation and join, probably the next most useful set of functions provided by the POSIX threads library are those for providing mutual exclusion to a critical section via **locks**. The most basic pair of routines to use for this purpose is provided by this pair of routines:

```

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

```

---

<sup>2</sup>Fortunately the compiler `gcc` will likely complain when you write code like this, which is yet another reason to pay attention to compiler warnings.

The routines should be easy to understand and use. When you have a region of code you realize is a **critical section**, and thus needs to be protected by locks in order to operate as desired. You can probably imagine what the code looks like:

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

The intent of the code is as follows: if no other thread holds the lock when `pthread_mutex_lock()` is called, the thread will acquire the lock and enter the critical section. If another thread does indeed hold the lock, the thread trying to grab the lock will not return from the call until it has acquired the lock (implying that the thread holding the lock has released it via the unlock call). Of course, many threads may be stuck waiting inside the lock acquisition function at a given time; only the thread with the lock acquired, however, should call unlock.

Unfortunately, this code is broken, in two important ways. The first problem is a **lack of proper initialization**. All locks must be properly initialized in order to guarantee that they have the correct values to begin with and thus work as desired when lock and unlock are called.

With POSIX threads, there are two ways to initialize locks. One way to do this is to use `PTHREAD_MUTEX_INITIALIZER`, as follows:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Doing so sets the lock to the default values and thus makes the lock usable. The dynamic way to do it (i.e., at run time) is to make a call to `pthread_mutex_init()`, as follows:

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

The first argument to this routine is the address of the lock itself, whereas the second is an optional set of attributes. Read more about the attributes yourself; passing `NULL` in simply uses the defaults. Either way works, but we usually use the dynamic (latter) method. Note that a corresponding call to `pthread_mutex_destroy()` should also be made, when you are done with the lock; see the manual page for all of details.

The second problem with the code above is that it fails to check error codes when calling lock and unlock. Just like virtually any library routine you call in a UNIX system, these routines can also fail! If your code doesn't properly check error codes, the failure will happen silently, which in this case could allow multiple threads into a critical section. Minimally, use wrappers, which assert that the routine succeeded (e.g., as in Figure 27.4); more sophisticated (non-toy) programs, which can't simply exit when something goes wrong, should check for failure and do something appropriate when the lock or unlock does not succeed.

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

Figure 27.4: An Example Wrapper

The lock and unlock routines are not the only routines within the pthreads library to interact with locks. In particular, here are two more routines which may be of interest:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                           struct timespec *abs_timeout);
```

These two calls are used in lock acquisition. The trylock version returns failure if the lock is already held; the timedlock version of acquiring a lock returns after a timeout or after acquiring the lock, whichever happens first. Thus, the timedlock with a timeout of zero degenerates to the trylock case. Both of these versions should generally be avoided; however, there are a few cases where avoiding getting stuck (perhaps indefinitely) in a lock acquisition routine can be useful, as we'll see in future chapters (e.g., when we study deadlock).

## 27.4 Condition Variables

The other major component of any threads library, and certainly the case with POSIX threads, is the presence of a **condition variable**. Condition variables are useful when some kind of signaling must take place between threads, if one thread is waiting for another to do something before it can continue. Two primary routines are used by programs wishing to interact in this way:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

To use a condition variable, one has to in addition have a lock that is associated with this condition. When calling either of the above routines, this lock should be held.

The first routine, `pthread_cond_wait()`, puts the calling thread to sleep, and thus waits for some other thread to signal it, usually when something in the program has changed that the now-sleeping thread might care about. A typical usage looks like this:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

In this code, after initialization of the relevant lock and condition<sup>3</sup>, a thread checks to see if the variable `ready` has yet been set to something other than zero. If not, the thread simply calls the wait routine in order to sleep until some other thread wakes it.

The code to wake a thread, which would run in some other thread, looks like this:

```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

A few things to note about this code sequence. First, when signaling (as well as when modifying the global variable `ready`), we always make sure to have the lock held. This ensures that we don't accidentally introduce a race condition into our code.

Second, you might notice that the wait call takes a lock as its second parameter, whereas the signal call only takes a condition. The reason for this difference is that the wait call, in addition to putting the calling thread to sleep, *releases* the lock when putting said caller to sleep. Imagine if it did not: how could the other thread acquire the lock and signal it to wake up? However, *before* returning after being woken, the `pthread_cond_wait()` re-acquires the lock, thus ensuring that any time the waiting thread is running between the lock acquire at the beginning of the wait sequence, and the lock release at the end, it holds the lock.

One last oddity: the waiting thread re-checks the condition in a while loop, instead of a simple if statement. We'll discuss this issue in detail when we study condition variables in a future chapter, but in general, using a while loop is the simple and safe thing to do. Although it rechecks the condition (perhaps adding a little overhead), there are some `pthread` implementations that could spuriously wake up a waiting thread; in such a case, without rechecking, the waiting thread will continue thinking that the condition has changed even though it has not. It is safer thus to view waking up as a hint that something might have changed, rather than an absolute fact.

Note that sometimes it is tempting to use a simple flag to signal between two threads, instead of a condition variable and associated lock. For example, we could rewrite the waiting code above to look more like this in the waiting code:

```
while (ready == 0)
    ; // spin
```

The associated signaling code would look like this:

```
ready = 1;
```

---

<sup>3</sup>Note that one could use `pthread_cond_init()` (and corresponding the `pthread_cond_destroy()` call) instead of the static initializer `PTHREAD_COND_INITIALIZER`. Sound like more work? It is.

Don't ever do this, for the following reasons. First, it performs poorly in many cases (spinning for a long time just wastes CPU cycles). Second, it is error prone. As recent research shows [X+10], it is surprisingly easy to make mistakes when using flags (as above) to synchronize between threads; in that study, roughly half the uses of these *ad hoc* synchronizations were buggy! Don't be lazy; use condition variables even when you think you can get away without doing so.

If condition variables sound confusing, don't worry too much (yet) – we'll be covering them in great detail in a subsequent chapter. Until then, it should suffice to know that they exist and to have some idea how and why they are used.

## 27.5 Compiling and Running

All of the code examples in this chapter are relatively easy to get up and running. To compile them, you must include the header `pthread.h` in your code. On the link line, you must also explicitly link with the `pthreads` library, by adding the `-pthread` flag.

For example, to compile a simple multi-threaded program, all you have to do is the following:

```
prompt> gcc -o main main.c -Wall -pthread
```

As long as `main.c` includes the `pthreads` header, you have now successfully compiled a concurrent program. Whether it works or not, as usual, is a different matter entirely.

## 27.6 Summary

We have introduced the basics of the `pthread` library, including thread creation, building mutual exclusion via locks, and signaling and waiting via condition variables. You don't need much else to write robust and efficient multi-threaded code, except patience and a great deal of care!

We now end the chapter with a set of tips that might be useful to you when you write multi-threaded code (see the aside on the following page for details). There are other aspects of the API that are interesting; if you want more information, type `man -k pthread` on a Linux system to see over one hundred APIs that make up the entire interface. However, the basics discussed herein should enable you to build sophisticated (and hopefully, correct and performant) multi-threaded programs. The hard part with threads is not the APIs, but rather the tricky logic of how you build concurrent programs. Read on to learn more.

**ASIDE: THREAD API GUIDELINES**

There are a number of small but important things to remember when you use the POSIX thread library (or really, any thread library) to build a multi-threaded program. They are:

- **Keep it simple.** Above all else, any code to lock or signal between threads should be as simple as possible. Tricky thread interactions lead to bugs.
- **Minimize thread interactions.** Try to keep the number of ways in which threads interact to a minimum. Each interaction should be carefully thought out and constructed with tried and true approaches (many of which we will learn about in the coming chapters).
- **Initialize locks and condition variables.** Failure to do so will lead to code that sometimes works and sometimes fails in very strange ways.
- **Check your return codes.** Of course, in any C and UNIX programming you do, you should be checking each and every return code, and it's true here as well. Failure to do so will lead to bizarre and hard to understand behavior, making you likely to (a) scream, (b) pull some of your hair out, or (c) both.
- **Be careful with how you pass arguments to, and return values from, threads.** In particular, any time you are passing a reference to a variable allocated on the stack, you are probably doing something wrong.
- **Each thread has its own stack.** As related to the point above, please remember that each thread has its own stack. Thus, if you have a locally-allocated variable inside of some function a thread is executing, it is essentially *private* to that thread; no other thread can (easily) access it. To share data between threads, the values must be in the **heap** or otherwise some locale that is globally accessible.
- **Always use condition variables to signal between threads.** While it is often tempting to use a simple flag, don't do it.
- **Use the manual pages.** On Linux, in particular, the pthread man pages are highly informative and discuss much of the nuances presented here, often in even more detail. Read them carefully!

## References

[B89] "An Introduction to Programming with Threads"

Andrew D. Birrell

DEC Technical Report, January, 1989

Available: <https://birrell.org/andrew/papers/035-Threads.pdf>

*A classic but older introduction to threaded programming. Still a worthwhile read, and freely available.*

[B97] "Programming with POSIX Threads"

David R. Butenhof

Addison-Wesley, May 1997

*Another one of these books on threads.*

[B+96] "PThreads Programming:

A POSIX Standard for Better Multiprocessing"

Dick Buttlar, Jacqueline Farrell, Bradford Nichols

O'Reilly, September 1996

*A reasonable book from the excellent, practical publishing house O'Reilly. Our bookshelves certainly contain a great deal of books from this company, including some excellent offerings on Perl, Python, and Javascript (particularly Crockford's "Javascript: The Good Parts").*

[K+96] "Programming With Threads"

Steve Kleiman, Devang Shah, Bart Smaalders

Prentice Hall, January 1996

*Probably one of the better books in this space. Get it at your local library. Or steal it from your mother. More seriously, just ask your mother for it – she'll let you borrow it, don't worry.*

[X+10] "Ad Hoc Synchronization Considered Harmful"

Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, Zhiqiang Ma

OSDI 2010, Vancouver, Canada

*This paper shows how seemingly simple synchronization code can lead to a surprising number of bugs. Use condition variables and do the signaling correctly!*

## Homework (Code)

In this section, we'll write some simple multi-threaded programs and use a specific tool, called **helgrind**, to find problems in these programs.

Read the README in the homework download for details on how to build the programs and run **helgrind**.

### Questions

1. First build `main-race.c`. Examine the code so you can see the (hopefully obvious) data race in the code. Now run `helgrind` (by typing `valgrind --tool=helgrind main-race`) to see how it reports the race. Does it point to the right lines of code? What other information does it give to you?
2. What happens when you remove one of the offending lines of code? Now add a lock around one of the updates to the shared variable, and then around both. What does `helgrind` report in each of these cases?
3. Now let's look at `main-deadlock.c`. Examine the code. This code has a problem known as **deadlock** (which we discuss in much more depth in a forthcoming chapter). Can you see what problem it might have?
4. Now run `helgrind` on this code. What does `helgrind` report?
5. Now run `helgrind` on `main-deadlock-global.c`. Examine the code; does it have the same problem that `main-deadlock.c` has? Should `helgrind` be reporting the same error? What does this tell you about tools like `helgrind`?
6. Let's next look at `main-signal.c`. This code uses a variable (`done`) to signal that the child is done and that the parent can now continue. Why is this code inefficient? (what does the parent end up spending its time doing, particularly if the child thread takes a long time to complete?)
7. Now run `helgrind` on this program. What does it report? Is the code correct?
8. Now look at a slightly modified version of the code, which is found in `main-signal-cv.c`. This version uses a condition variable to do the signaling (and associated lock). Why is this code preferred to the previous version? Is it correctness, or performance, or both?
9. Once again run `helgrind` on `main-signal-cv`. Does it report any errors?

# User-Mode Thread Implementation

## Introduction

For a long time, processes were the only unit of parallel computation. There are two problems with this:

- processes are very expensive to create and dispatch, due to the fact that each has its own virtual address space and ownership of numerous system resources.
- each process operates in its own address space, and cannot share in-memory resources with parallel processes (this was before it was possible to map shared segments into a process' address space).

The answer to these needs was to create threads. A thread ...

- is an independently schedulable unit of execution.
- runs within the address space of a process.
- has access to all of the system resources owned by that process.
- has its own general registers.
- has its own stack (within the owning process' address space).

Threads were added to Unix/Linux as an after-thought. In Unix, a process could be scheduled for execution, and could create threads for additional parallelism. Windows NT is a newer operating system, and it was designed with threads from the start ... and so the abstractions are cleaner:

- a process is a container for an address space and resources.
- a thread is **the** unit of scheduled execution.

## A Simple Threads Library

When threads were first added to Linux they were entirely implemented in a user-mode library, with no assistance from the operating system. This is not merely historical trivia, but interesting as an examination of what kinds of problems can and cannot be solved without operating system assistance.

The basic model is:

- Each time a new thread was created:
  - we allocate memory for a (fixed size) thread-private stack from the heap.
  - we create a new thread descriptor that contains identification information, scheduling information, and a pointer to the stack.
  - we add the new thread to a ready queue.
- When a thread calls *yield()* or *sleep()* we save its general registers (on its own stack), and then select the next thread on the ready queue.
- To dispatch a new thread, we simply restore its saved registers (including the stack pointer), and return from the call that caused it to *yield*.
- If a thread called *sleep()* we would remove it from the ready queue. When it was re-awakened, we would put it back onto the ready queue.
- When a thread exited, we would free its stack and thread descriptor.

Eventually people wanted preemptive scheduling to ensure good interactive response and prevent buggy threads from tying up the application:

- Linux processes can schedule the delivery of (SIGALARM) timer signals and register a handler for them.

- Before dispatching a thread, the we can schedule a SIGALARM that will interrupt the thread if it runs too long.
- If a thread runs too long, the SIGALARM handler can *yield* on behalf of that thread, saving its state, moving on to the next thread in the ready queue.

But the addition of preemptive scheduling created new problems for critical sections that required before-or-after, all-or-none serialization. Fortunately Linux processes can temporarily block signals (much as it is possible to temporarily disable an interrupt) via the *sigprocmask(2)* system call.

## Kernel implemented threads

There are two fundamental problems with implementing threads in a user mode library:

- what happens when a system call blocks

If a user-mode thread issues a system call that blocks (e.g. *open* or *read*), the process is blocked until that operation completes. This means that when a thread blocks, all threads (within that process) stop executing. Since the threads were implemented in user-mode, the operating system has no knowledge of them, and cannot know that other threads (in that process) might still be runnable.

- exploiting multi-processors

If the CPU has multiple execution cores, the operating system can schedule processes on each to run in parallel. But if the operating system is not aware that a process is comprised of multiple threads, those threads cannot execute in parallel on the available cores.

Both of these problems are solved if threads are implemented by the operating system rather than by a user-mode thread library.

## Performance Implications

If non-preemptive scheduling can be used, user-mode threads operating in with a *sleep/yield* model are much more efficient than doing context switches through the operating system. There are, today, *light weight thread* implementations to reap these benefits.

If preemptive scheduling is to be used, the costs of setting alarms and servicing the signals may well be greater than the cost of simply allowing the operating system to do the scheduling.

If the threads can run in parallel on a multi-processor, the added throughput resulting from true parallel execution may be far greater than the efficiency losses associated with more expensive context switches through the operating system. Also, the operating system knows which threads are part of the same process, and may be able to schedule them to maximize cache-line sharing.

Like preemptive scheduling, the signal disabling and reenabling for a user-mode mutex or condition variable implementation may be more expensive than simply using the kernel-mode implementations. But it may be possible to get the best of both worlds with a user-mode implementation that uses an atomic instruction to attempt to get a lock, but calls the operating system if that allocation fails (somewhat like the *futex(7)* approach).



## Locks

From the introduction to concurrency, we saw one of the fundamental problems in concurrent programming: we would like to execute a series of instructions atomically, but due to the presence of interrupts on a single processor (or multiple threads executing on multiple processors concurrently), we couldn't. In this chapter, we thus attack this problem directly, with the introduction of something referred to as a **lock**. Programmers annotate source code with locks, putting them around critical sections, and thus ensure that any such critical section executes as if it were a single atomic instruction.

### 28.1 Locks: The Basic Idea

As an example, assume our critical section looks like this, the canonical update of a shared variable:

```
balance = balance + 1;
```

Of course, other critical sections are possible, such as adding an element to a linked list or other more complex updates to shared structures, but we'll just keep to this simple example for now. To use a lock, we add some code around the critical section like this:

```
1 lock_t mutex; // some globally-allocated lock 'mutex'
2 ...
3 lock(&mutex);
4 balance = balance + 1;
5 unlock(&mutex);
```

A lock is just a variable, and thus to use one, you must declare a **lock variable** of some kind (such as `mutex` above). This lock variable (or just "lock" for short) holds the state of the lock at any instant in time. It is either **available** (or **unlocked** or **free**) and thus no thread holds the lock, or **acquired** (or **locked** or **held**), and thus exactly one thread holds the lock and presumably is in a critical section. We could store other information

in the data type as well, such as which thread holds the lock, or a queue for ordering lock acquisition, but information like that is hidden from the user of the lock.

The semantics of the `lock()` and `unlock()` routines are simple. Calling the routine `lock()` tries to acquire the lock; if no other thread holds the lock (i.e., it is free), the thread will acquire the lock and enter the critical section; this thread is sometimes said to be the **owner** of the lock. If another thread then calls `lock()` on that same lock variable (`mutex` in this example), it will not return while the lock is held by another thread; in this way, other threads are prevented from entering the critical section while the first thread that holds the lock is in there.

Once the owner of the lock calls `unlock()`, the lock is now available (free) again. If no other threads are waiting for the lock (i.e., no other thread has called `lock()` and is stuck therein), the state of the lock is simply changed to free. If there are waiting threads (stuck in `lock()`), one of them will (eventually) notice (or be informed of) this change of the lock's state, acquire the lock, and enter the critical section.

Locks provide some minimal amount of control over scheduling to programmers. In general, we view threads as entities created by the programmer but scheduled by the OS, in any fashion that the OS chooses. Locks yield some of that control back to the programmer; by putting a lock around a section of code, the programmer can guarantee that no more than a single thread can ever be active within that code. Thus locks help transform the chaos that is traditional OS scheduling into a more controlled activity.

## 28.2 Pthread Locks

The name that the POSIX library uses for a lock is a **mutex**, as it is used to provide **mutual exclusion** between threads, i.e., if one thread is in the critical section, it excludes the others from entering until it has completed the section. Thus, when you see the following POSIX threads code, you should understand that it is doing the same thing as above (we again use our wrappers that check for errors upon lock and unlock):

```

1  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2
3  Pthread_mutex_lock(&lock);    // wrapper for pthread_mutex_lock()
4  balance = balance + 1;
5  Pthread_mutex_unlock(&lock);

```

You might also notice here that the POSIX version passes a variable to lock and unlock, as we may be using *different* locks to protect different variables. Doing so can increase concurrency: instead of one big lock that is used any time any critical section is accessed (a **coarse-grained** locking strategy), one will often protect different data and data structures with different locks, thus allowing more threads to be in locked code at once (a more **fine-grained** approach).

## 28.3 Building A Lock

By now, you should have some understanding of how a lock works, from the perspective of a programmer. But how should we build a lock? What hardware support is needed? What OS support? It is this set of questions we address in the rest of this chapter.

### The Crux: How To BUILD A LOCK

How can we build an efficient lock? Efficient locks provided mutual exclusion at low cost, and also might attain a few other properties we discuss below. What hardware support is needed? What OS support?

To build a working lock, we will need some help from our old friend, the hardware, as well as our good pal, the OS. Over the years, a number of different hardware primitives have been added to the instruction sets of various computer architectures; while we won't study how these instructions are implemented (that, after all, is the topic of a computer architecture class), we will study how to use them in order to build a mutual exclusion primitive like a lock. We will also study how the OS gets involved to complete the picture and enable us to build a sophisticated locking library.

## 28.4 Evaluating Locks

Before building any locks, we should first understand what our goals are, and thus we ask how to evaluate the efficacy of a particular lock implementation. To evaluate whether a lock works (and works well), we should first establish some basic criteria. The first is whether the lock does its basic task, which is to provide **mutual exclusion**. Basically, does the lock work, preventing multiple threads from entering a critical section?

The second is **fairness**. Does each thread contending for the lock get a fair shot at acquiring it once it is free? Another way to look at this is by examining the more extreme case: does any thread contending for the lock **starve** while doing so, thus never obtaining it?

The final criterion is **performance**, specifically the time overheads added by using the lock. There are a few different cases that are worth considering here. One is the case of no contention; when a single thread is running and grabs and releases the lock, what is the overhead of doing so? Another is the case where multiple threads are contending for the lock on a single CPU; in this case, are there performance concerns? Finally, how does the lock perform when there are multiple CPUs involved, and threads on each contending for the lock? By comparing these different scenarios, we can better understand the performance impact of using various locking techniques, as described below.

## 28.5 Controlling Interrupts

One of the earliest solutions used to provide mutual exclusion was to disable interrupts for critical sections; this solution was invented for single-processor systems. The code would look like this:

```

1 void lock() {
2     DisableInterrupts();
3 }
4 void unlock() {
5     EnableInterrupts();
6 }
```

Assume we are running on such a single-processor system. By turning off interrupts (using some kind of special hardware instruction) before entering a critical section, we ensure that the code inside the critical section will *not* be interrupted, and thus will execute as if it were atomic. When we are finished, we re-enable interrupts (again, via a hardware instruction) and thus the program proceeds as usual.

The main positive of this approach is its simplicity. You certainly don't have to scratch your head too hard to figure out why this works. Without interruption, a thread can be sure that the code it executes will execute and that no other thread will interfere with it.

The negatives, unfortunately, are many. First, this approach requires us to allow any calling thread to perform a *privileged* operation (turning interrupts on and off), and thus *trust* that this facility is not abused. As you already know, any time we are required to trust an arbitrary program, we are probably in trouble. Here, the trouble manifests in numerous ways: a greedy program could call `lock()` at the beginning of its execution and thus monopolize the processor; worse, an errant or malicious program could call `lock()` and go into an endless loop. In this latter case, the OS never regains control of the system, and there is only one recourse: restart the system. Using interrupt disabling as a general-purpose synchronization solution requires too much trust in applications.

Second, the approach does not work on multiprocessors. If multiple threads are running on different CPUs, and each try to enter the same critical section, it does not matter whether interrupts are disabled; threads will be able to run on other processors, and thus could enter the critical section. As multiprocessors are now commonplace, our general solution will have to do better than this.

Third, turning off interrupts for extended periods of time can lead to interrupts becoming lost, which can lead to serious systems problems. Imagine, for example, if the CPU missed the fact that a disk device has finished a read request. How will the OS know to wake the process waiting for said read?

Finally, and probably least important, this approach can be inefficient. Compared to normal instruction execution, code that masks or unmasks interrupts tends to be executed slowly by modern CPUs.

For these reasons, turning off interrupts is only used in limited contexts as a mutual-exclusion primitive. For example, in some cases an

### ASIDE: DEKKER'S AND PETERSON'S ALGORITHMS

In the 1960's, Dijkstra posed the concurrency problem to his friends, and one of them, a mathematician named Theodorus Jozef Dekker, came up with a solution [D68]. Unlike the solutions we discuss here, which use special hardware instructions and even OS support, **Dekker's algorithm** uses just loads and stores (assuming they are atomic with respect to each other, which was true on early hardware).

Dekker's approach was later refined by Peterson [P81]. Once again, just loads and stores are used, and the idea is to ensure that two threads never enter a critical section at the same time. Here is **Peterson's algorithm** (for two threads); see if you can understand the code. What are the `flag` and `turn` variables used for?

```
int flag[2];
int turn;

void init() {
    flag[0] = flag[1] = 0;      // 1->thread wants to grab lock
    turn = 0;                  // whose turn? (thread 0 or 1?)
}
void lock() {
    flag[self] = 1;            // self: thread ID of caller
    turn = 1 - self;          // make it other thread's turn
    while ((flag[1-self] == 1) && (turn == 1 - self))
        ; // spin-wait
}
void unlock() {
    flag[self] = 0;            // simply undo your intent
}
```

For some reason, developing locks that work without special hardware support became all the rage for a while, giving theory-types a lot of problems to work on. Of course, this line of work became quite useless when people realized it is much easier to assume a little hardware support (and indeed that support had been around from the earliest days of multiprocessing). Further, algorithms like the ones above don't work on modern hardware (due to relaxed memory consistency models), thus making them even less useful than they were before. Yet more research relegated to the dustbin of history...

operating system itself will use interrupt masking to guarantee atomicity when accessing its own data structures, or at least to prevent certain messy interrupt handling situations from arising. This usage makes sense, as the trust issue disappears inside the OS, which always trusts itself to perform privileged operations anyhow.

```

1  typedef struct __lock_t { int flag; } lock_t;
2
3  void init(lock_t *mutex) {
4      // 0 -> lock is available, 1 -> held
5      mutex->flag = 0;
6  }
7
8  void lock(lock_t *mutex) {
9      while (mutex->flag == 1) // TEST the flag
10         ; // spin-wait (do nothing)
11     mutex->flag = 1;           // now SET it!
12 }
13
14 void unlock(lock_t *mutex) {
15     mutex->flag = 0;
16 }
```

Figure 28.1: First Attempt: A Simple Flag

## 28.6 Test And Set (Atomic Exchange)

Because disabling interrupts does not work on multiple processors, system designers started to invent hardware support for locking. The earliest multiprocessor systems, such as the Burroughs B5000 in the early 1960's [M82], had such support; today all systems provide this type of support, even for single CPU systems.

The simplest bit of hardware support to understand is what is known as a **test-and-set instruction**, also known as **atomic exchange**. To understand how test-and-set works, let's first try to build a simple lock without it. In this failed attempt, we use a simple flag variable to denote whether the lock is held or not.

In this first attempt (Figure 28.1), the idea is quite simple: use a simple variable to indicate whether some thread has possession of a lock. The first thread that enters the critical section will call `lock()`, which **tests** whether the flag is equal to 1 (in this case, it is not), and then **sets** the flag to 1 to indicate that the thread now **holds** the lock. When finished with the critical section, the thread calls `unlock()` and clears the flag, thus indicating that the lock is no longer held.

If another thread happens to call `lock()` while that first thread is in the critical section, it will simply **spin-wait** in the while loop for that thread to call `unlock()` and clear the flag. Once that first thread does so, the waiting thread will fall out of the while loop, set the flag to 1 for itself, and proceed into the critical section.

Unfortunately, the code has two problems: one of correctness, and another of performance. The correctness problem is simple to see once you get used to thinking about concurrent programming. Imagine the code interleaving in Figure 28.2 (page 7); assume `flag=0` to begin.

As you can see from this interleaving, with timely (untimely?) interrupts, we can easily produce a case where *both* threads set the flag to 1 and both threads are thus able to enter the critical section. This behavior is what professionals call "bad" – we have obviously failed to provide the most basic requirement: providing mutual exclusion.

Thread 1	Thread 2
call lock() while (flag == 1) <b>interrupt: switch to Thread 2</b>  <b>flag = 1; // set flag to 1 (too!)</b>	call lock() while (flag == 1) flag = 1; <b>interrupt: switch to Thread 1</b>

Figure 28.2: Trace: No Mutual Exclusion

The performance problem, which we will address more later on, is the fact that the way a thread waits to acquire a lock that is already held: it endlessly checks the value of flag, a technique known as **spin-waiting**. Spin-waiting wastes time waiting for another thread to release a lock. The waste is exceptionally high on a uniprocessor, where the thread that the waiter is waiting for cannot even run (at least, until a context switch occurs)! Thus, as we move forward and develop more sophisticated solutions, we should also consider ways to avoid this kind of waste.

## 28.7 Building A Working Spin Lock

While the idea behind the example above is a good one, it is not possible to implement without some support from the hardware. Fortunately, some systems provide an instruction to support the creation of simple locks based on this concept. This more powerful instruction has different names — on SPARC, it is the load/store unsigned byte instruction (`ldstub`), whereas on x86, it is the atomic exchange instruction (`xchng`) — but basically does the same thing across platforms, and is generally referred to as **test-and-set**. We define what the test-and-set instruction does with the following C code snippet:

```

1 int TestAndSet(int *old_ptr, int new) {
2     int old = *old_ptr; // fetch old value at old_ptr
3     *old_ptr = new;    // store 'new' into old_ptr
4     return old;        // return the old value
5 }
```

What the test-and-set instruction does is as follows. It returns the old value pointed to by the `ptr`, and simultaneously updates said value to new. The key, of course, is that this sequence of operations is performed **atomically**. The reason it is called “test and set” is that it enables you to “test” the old value (which is what is returned) while simultaneously “setting” the memory location to a new value; as it turns out, this slightly more powerful instruction is enough to build a simple **spin lock**, as we now examine in Figure 28.3. Or better yet: figure it out first yourself!

Let’s make sure we understand why this lock works. Imagine first the case where a thread calls `lock()` and no other thread currently holds the lock; thus, `flag` should be 0. When the thread calls `TestAndSet(flag,`

```

1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0 indicates that lock is available, 1 that it is held
7      lock->flag = 0;
8  }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

Figure 28.3: A Simple Spin Lock Using Test-and-set

1), the routine will return the old value of `flag`, which is 0; thus, the calling thread, which is *testing* the value of `flag`, will not get caught spinning in the while loop and will acquire the lock. The thread will also atomically *set* the value to 1, thus indicating that the lock is now held. When the thread is finished with its critical section, it calls `unlock()` to set the `flag` back to zero.

The second case we can imagine arises when one thread already has the lock held (i.e., `flag` is 1). In this case, this thread will call `lock()` and then call `TestAndSet(flag, 1)` as well. This time, `TestAndSet()` will return the old value at `flag`, which is 1 (because the lock is held), while simultaneously setting it to 1 again. As long as the lock is held by another thread, `TestAndSet()` will repeatedly return 1, and thus this thread will spin and spin until the lock is finally released. When the `flag` is finally set to 0 by some other thread, this thread will call `TestAndSet()` again, which will now return 0 while atomically setting the value to 1 and thus acquire the lock and enter the critical section.

By making both the **test** (of the old lock value) and **set** (of the new value) a single atomic operation, we ensure that only one thread acquires the lock. And that's how to build a working mutual exclusion primitive!

You may also now understand why this type of lock is usually referred

#### TIP: THINK ABOUT CONCURRENCY AS MALICIOUS SCHEDULER

From this example, you might get a sense of the approach you need to take to understand concurrent execution. What you should try to do is to pretend you are a **malicious scheduler**, one that interrupts threads at the most inopportune of times in order to foil their feeble attempts at building synchronization primitives. What a mean scheduler you are! Although the exact sequence of interrupts may be *improbable*, it is *possible*, and that is all we need to demonstrate that a particular approach does not work. It can be useful to think maliciously! (at least, sometimes)

to as a **spin lock**. It is the simplest type of lock to build, and simply spins, using CPU cycles, until the lock becomes available. To work correctly on a single processor, it requires a **preemptive scheduler** (i.e., one that will interrupt a thread via a timer, in order to run a different thread, from time to time). Without preemption, spin locks don't make much sense on a single CPU, as a thread spinning on a CPU will never relinquish it.

## 28.8 Evaluating Spin Locks

Given our basic spin lock, we can now evaluate how effective it is along our previously described axes. The most important aspect of a lock is **correctness**: does it provide mutual exclusion? The answer here is yes: the spin lock only allows a single thread to enter the critical section at a time. Thus, we have a correct lock.

The next axis is **fairness**. How fair is a spin lock to a waiting thread? Can you guarantee that a waiting thread will ever enter the critical section? The answer here, unfortunately, is bad news: spin locks don't provide any fairness guarantees. Indeed, a thread spinning may spin forever, under contention. Spin locks are not fair and may lead to starvation.

The final axis is **performance**. What are the costs of using a spin lock? To analyze this more carefully, we suggest thinking about a few different cases. In the first, imagine threads competing for the lock on a single processor; in the second, consider the threads as spread out across many processors.

For spin locks, in the single CPU case, performance overheads can be quite painful; imagine the case where the thread holding the lock is pre-empted within a critical section. The scheduler might then run every other thread (imagine there are  $N - 1$  others), each of which tries to acquire the lock. In this case, each of those threads will spin for the duration of a time slice before giving up the CPU, a waste of CPU cycles.

However, on multiple CPUs, spin locks work reasonably well (if the number of threads roughly equals the number of CPUs). The thinking goes as follows: imagine Thread A on CPU 1 and Thread B on CPU 2, both contending for a lock. If Thread A (CPU 1) grabs the lock, and then Thread B tries to, B will spin (on CPU 2). However, presumably the critical section is short, and thus soon the lock becomes available, and is acquired by Thread B. Spinning to wait for a lock held on another processor doesn't waste many cycles in this case, and thus can be effective.

## 28.9 Compare-And-Swap

Another hardware primitive that some systems provide is known as the **compare-and-swap** instruction (as it is called on SPARC, for example), or **compare-and-exchange** (as it called on x86). The C pseudocode for this single instruction is found in Figure 28.4.

The basic idea is for compare-and-swap to test whether the value at the

```

1 int CompareAndSwap(int *ptr, int expected, int new) {
2     int actual = *ptr;
3     if (actual == expected)
4         *ptr = new;
5     return actual;
6 }
```

Figure 28.4: Compare-and-swap

address specified by `ptr` is equal to `expected`; if so, update the memory location pointed to by `ptr` with the new value. If not, do nothing. In either case, return the actual value at that memory location, thus allowing the code calling compare-and-swap to know whether it succeeded or not.

With the compare-and-swap instruction, we can build a lock in a manner quite similar to that with test-and-set. For example, we could just replace the `lock()` routine above with the following:

```

1 void lock(lock_t *lock) {
2     while (CompareAndSwap(&lock->flag, 0, 1) == 1)
3         ; // spin
4 }
```

The rest of the code is the same as the test-and-set example above. This code works quite similarly; it simply checks if the flag is 0 and if so, atomically swaps in a 1 thus acquiring the lock. Threads that try to acquire the lock while it is held will get stuck spinning until the lock is finally released.

If you want to see how to really make a C-callable x86-version of compare-and-swap, this code sequence might be useful (from [S05]):

```

1 char CompareAndSwap(int *ptr, int old, int new) {
2     unsigned char ret;
3
4     // Note that sete sets a 'byte' not the word
5     __asm__ __volatile__ (
6         "lock\n"
7         "cmpxchgl %2,%1\n"
8         "sete %0\n"
9         : "=q" (ret), "=m" (*ptr)
10        : "r" (new), "m" (*ptr), "a" (old)
11        : "memory");
12     return ret;
13 }
```

Finally, as you may have sensed, compare-and-swap is a more powerful instruction than test-and-set. We will make some use of this power in the future when we briefly delve into topics such as **lock-free synchronization** [H91]. However, if we just build a simple spin lock with it, its behavior is identical to the spin lock we analyzed above.

```

1 int LoadLinked(int *ptr) {
2     return *ptr;
3 }
4
5 int StoreConditional(int *ptr, int value) {
6     if (no one has updated *ptr since the LoadLinked to this address) {
7         *ptr = value;
8         return 1; // success!
9     } else {
10        return 0; // failed to update
11    }
12 }
```

Figure 28.5: Load-linked And Store-conditional

## 28.10 Load-Linked and Store-Conditional

Some platforms provide a pair of instructions that work in concert to help build critical sections. On the MIPS architecture [H93], for example, the **load-linked** and **store-conditional** instructions can be used in tandem to build locks and other concurrent structures. The C pseudocode for these instructions is as found in Figure 28.5. Alpha, PowerPC, and ARM provide similar instructions [W09].

The load-linked operates much like a typical load instruction, and simply fetches a value from memory and places it in a register. The key difference comes with the store-conditional, which only succeeds (and updates the value stored at the address just load-linked from) if no intervening store to the address has taken place. In the case of success, the store-conditional returns 1 and updates the value at `ptr` to `value`; if it fails, the value at `ptr` is *not* updated and 0 is returned.

As a challenge to yourself, try thinking about how to build a lock using load-linked and store-conditional. Then, when you are finished, look at the code below which provides one simple solution. Do it! The solution is in Figure 28.6.

The `lock()` code is the only interesting piece. First, a thread spins waiting for the flag to be set to 0 (and thus indicate the lock is not held). Once so, the thread tries to acquire the lock via the store-conditional; if it succeeds, the thread has atomically changed the flag's value to 1 and thus can proceed into the critical section.

```

1 void lock(lock_t *lock) {
2     while (1) {
3         while (LoadLinked(&lock->flag) == 1)
4             ; // spin until it's zero
5         if (StoreConditional(&lock->flag, 1) == 1)
6             return; // if set-it-to-1 was a success: all done
7             // otherwise: try it all over again
8     }
9 }
10
11 void unlock(lock_t *lock) {
12     lock->flag = 0;
13 }
```

Figure 28.6: Using LL/SC To Build A Lock

## TIP: LESS CODE IS BETTER CODE (LAUER'S LAW)

Programmers tend to brag about how much code they wrote to do something. Doing so is fundamentally broken. What one should brag about, rather, is how *little* code one wrote to accomplish a given task. Short, concise code is always preferred; it is likely easier to understand and has fewer bugs. As Hugh Lauer said, when discussing the construction of the Pilot operating system: “If the same people had twice as much time, they could produce as good of a system in half the code.” [L81] We’ll call this **Lauer’s Law**, and it is well worth remembering. So next time you’re bragging about how much code you wrote to finish the assignment, think again, or better yet, go back, rewrite, and make the code as clear and concise as possible.

Note how failure of the store-conditional might arise. One thread calls `lock()` and executes the load-linked, returning 0 as the lock is not held. Before it can attempt the store-conditional, it is interrupted and another thread enters the lock code, also executing the load-linked instruction, and also getting a 0 and continuing. At this point, two threads have each executed the load-linked and each are about to attempt the store-conditional. The key feature of these instructions is that only one of these threads will succeed in updating the flag to 1 and thus acquire the lock; the second thread to attempt the store-conditional will fail (because the other thread updated the value of flag between its load-linked and store-conditional) and thus have to try to acquire the lock again.

In class a few years ago, undergraduate student David Capel suggested a more concise form of the above, for those of you who enjoy short-circuiting boolean conditionals. See if you can figure out why it is equivalent. It certainly is shorter!

```

1 void lock(lock_t *lock) {
2     while (LoadLinked(&lock->flag) || !StoreConditional(&lock->flag, 1))
3         ; // spin
4 }
```

## 28.11 Fetch-And-Add

One final hardware primitive is the **fetch-and-add** instruction, which atomically increments a value while returning the old value at a particular address. The C pseudocode for the fetch-and-add instruction looks like this:

```

1 int FetchAndAdd(int *ptr) {
2     int old = *ptr;
3     *ptr = old + 1;
4     return old;
5 }
```

```

1  typedef struct __lock_t {
2      int ticket;
3      int turn;
4  } lock_t;
5
6  void lock_init(lock_t *lock) {
7      lock->ticket = 0;
8      lock->turn   = 0;
9  }
10
11 void lock(lock_t *lock) {
12     int myturn = FetchAndAdd(&lock->ticket);
13     while (lock->turn != myturn)
14         ; // spin
15 }
16
17 void unlock(lock_t *lock) {
18     lock->turn = lock->turn + 1;
19 }
```

Figure 28.7: Ticket Locks

In this example, we'll use fetch-and-add to build a more interesting **ticket lock**, as introduced by Mellor-Crummey and Scott [MS91]. The lock and unlock code looks like what you see in Figure 28.7.

Instead of a single value, this solution uses a ticket and turn variable in combination to build a lock. The basic operation is pretty simple: when a thread wishes to acquire a lock, it first does an atomic fetch-and-add on the ticket value; that value is now considered this thread's "turn" (`myturn`). The globally shared `lock->turn` is then used to determine which thread's turn it is; when (`myturn == turn`) for a given thread, it is that thread's turn to enter the critical section. Unlock is accomplished simply by incrementing the turn such that the next waiting thread (if there is one) can now enter the critical section.

Note one important difference with this solution versus our previous attempts: it ensures progress for all threads. Once a thread is assigned its ticket value, it will be scheduled at some point in the future (once those in front of it have passed through the critical section and released the lock). In our previous attempts, no such guarantee existed; a thread spinning on test-and-set (for example) could spin forever even as other threads acquire and release the lock.

## 28.12 Too Much Spinning: What Now?

Our simple hardware-based locks are simple (only a few lines of code) and they work (you could even prove that if you'd like to, by writing some code), which are two excellent properties of any system or code. However, in some cases, these solutions can be quite inefficient. Imagine you are running two threads on a single processor. Now imagine that one thread (thread 0) is in a critical section and thus has a lock held, and unfortunately gets interrupted. The second thread (thread 1) now tries to acquire the lock, but finds that it is held. Thus, it begins to spin. And spin.

Then it spins some more. And finally, a timer interrupt goes off, thread 0 is run again, which releases the lock, and finally (the next time it runs, say), thread 1 won't have to spin so much and will be able to acquire the lock. Thus, any time a thread gets caught spinning in a situation like this, it wastes an entire time slice doing nothing but checking a value that isn't going to change! The problem gets worse with  $N$  threads contending for a lock;  $N - 1$  time slices may be wasted in a similar manner, simply spinning and waiting for a single thread to release the lock. And thus, our next problem:

#### THE CRUX: HOW TO AVOID SPINNING

How can we develop a lock that doesn't needlessly waste time spinning on the CPU?

Hardware support alone cannot solve the problem. We'll need OS support too! Let's now figure out just how that might work.

### 28.13 A Simple Approach: Just Yield, Baby

Hardware support got us pretty far: working locks, and even (as with the case of the ticket lock) fairness in lock acquisition. However, we still have a problem: what to do when a context switch occurs in a critical section, and threads start to spin endlessly, waiting for the interrupted (lock-holding) thread to be run again?

Our first try is a simple and friendly approach: when you are going to spin, instead give up the CPU to another thread. Or, as Al Davis might say, "just yield, baby!" [D91]. Figure 28.8 presents the approach.

In this approach, we assume an operating system primitive `yield()` which a thread can call when it wants to give up the CPU and let another thread run. A thread can be in one of three states (running, ready, or blocked); `yield` is simply a system call that moves the caller from the

```

1 void init() {
2     flag = 0;
3 }
4
5 void lock() {
6     while (TestAndSet(&flag, 1) == 1)
7         yield(); // give up the CPU
8 }
9
10 void unlock() {
11     flag = 0;
12 }
```

Figure 28.8: Lock With Test-and-set And Yield

**running** state to the **ready** state, and thus promotes another thread to running. Thus, the yielding process essentially **deschedules** itself.

Think about the example with two threads on one CPU; in this case, our yield-based approach works quite well. If a thread happens to call `lock()` and find a lock held, it will simply yield the CPU, and thus the other thread will run and finish its critical section. In this simple case, the yielding approach works well.

Let us now consider the case where there are many threads (say 100) contending for a lock repeatedly. In this case, if one thread acquires the lock and is preempted before releasing it, the other 99 will each call `lock()`, find the lock held, and yield the CPU. Assuming some kind of round-robin scheduler, each of the 99 will execute this run-and-yield pattern before the thread holding the lock gets to run again. While better than our spinning approach (which would waste 99 time slices spinning), this approach is still costly; the cost of a context switch can be substantial, and there is thus plenty of waste.

Worse, we have not tackled the starvation problem at all. A thread may get caught in an endless yield loop while other threads repeatedly enter and exit the critical section. We clearly will need an approach that addresses this problem directly.

## 28.14 Using Queues: Sleeping Instead Of Spinning

The real problem with our previous approaches is that they leave too much to chance. The scheduler determines which thread runs next; if the scheduler makes a bad choice, a thread runs that must either spin waiting for the lock (our first approach), or yield the CPU immediately (our second approach). Either way, there is potential for waste and no prevention of starvation.

Thus, we must explicitly exert some control over which thread next gets to acquire the lock after the current holder releases it. To do this, we will need a little more OS support, as well as a queue to keep track of which threads are waiting to acquire the lock.

For simplicity, we will use the support provided by Solaris, in terms of two calls: `park()` to put a calling thread to sleep, and `unpark(threadID)` to wake a particular thread as designated by `threadID`. These two routines can be used in tandem to build a lock that puts a caller to sleep if it tries to acquire a held lock and wakes it when the lock is free. Let's look at the code in Figure 28.9 to understand one possible use of such primitives.

We do a couple of interesting things in this example. First, we combine the old test-and-set idea with an explicit queue of lock waiters to make a more efficient lock. Second, we use a queue to help control who gets the lock next and thus avoid starvation.

You might notice how the guard is used (Figure 28.9, page 16), basically as a spin-lock around the flag and queue manipulations the lock is using. This approach thus doesn't avoid spin-waiting entirely; a thread

```

1  typedef struct __lock_t {
2      int flag;
3      int guard;
4      queue_t *q;
5  } lock_t;
6
7  void lock_init(lock_t *m) {
8      m->flag = 0;
9      m->guard = 0;
10     queue_init(m->q);
11 }
12
13 void lock(lock_t *m) {
14     while (TestAndSet(&m->guard, 1) == 1)
15         ; //acquire guard lock by spinning
16     if (m->flag == 0) {
17         m->flag = 1; // lock is acquired
18         m->guard = 0;
19     } else {
20         queue_add(m->q, gettid());
21         m->guard = 0;
22         park();
23     }
24 }
25
26 void unlock(lock_t *m) {
27     while (TestAndSet(&m->guard, 1) == 1)
28         ; //acquire guard lock by spinning
29     if (queue_empty(m->q))
30         m->flag = 0; // let go of lock; no one wants it
31     else
32         unpark(queue_remove(m->q)); // hold lock (for next thread!)
33     m->guard = 0;
34 }
```

Figure 28.9: Lock With Queues, Test-and-set, Yield, And Wakeup

might be interrupted while acquiring or releasing the lock, and thus cause other threads to spin-wait for this one to run again. However, the time spent spinning is quite limited (just a few instructions inside the lock and unlock code, instead of the user-defined critical section), and thus this approach may be reasonable.

Second, you might notice that in `lock()`, when a thread can not acquire the lock (it is already held), we are careful to add ourselves to a queue (by calling the `gettid()` call to get the thread ID of the current thread), set guard to 0, and yield the CPU. A question for the reader: What would happen if the release of the guard lock came *after* the `park()`, and not before? Hint: something bad.

You might also notice the interesting fact that the flag does not get set back to 0 when another thread gets woken up. Why is this? Well, it is not an error, but rather a necessity! When a thread is woken up, it will be as if it is returning from `park()`; however, it does not hold the guard at that point in the code and thus cannot even try to set the flag to 1. Thus, we just pass the lock directly from the thread releasing the lock to the next thread acquiring it; flag is not set to 0 in-between.

**ASIDE: MORE REASON TO AVOID SPINNING: PRIORITY INVERSION**

One good reason to avoid spin locks is performance: as described in the main text, if a thread is interrupted while holding a lock, other threads that use spin locks will spend a large amount of CPU time just waiting for the lock to become available. However, it turns out there is another interesting reason to avoid spin locks on some systems: correctness. The problem to be wary of is known as **priority inversion**, which unfortunately is an intergalactic scourge, occurring on Earth [M15] and Mars [R97]!

Let's assume there are two threads in a system. Thread 2 (T2) has a high scheduling priority, and Thread 1 (T1) has lower priority. In this example, let's assume that the CPU scheduler will always run T2 over T1, if indeed both are runnable; T1 only runs when T2 is not able to do so (e.g., when T2 is blocked on I/O).

Now, the problem. Assume T2 is blocked for some reason. So T1 runs, grabs a spin lock, and enters a critical section. T2 now becomes unblocked (perhaps because an I/O completed), and the CPU scheduler immediately schedules it (thus descheduling T1). T2 now tries to acquire the lock, and because it can't (T1 holds the lock), it just keeps spinning. Because the lock is a spin lock, T2 spins forever, and the system is hung.

Just avoiding the use of spin locks, unfortunately, does not avoid the problem of inversion (alas). Imagine three threads, T1, T2, and T3, with T3 at the highest priority, and T1 the lowest. Imagine now that T1 grabs a lock. T3 then starts, and because it is higher priority than T1, runs immediately (preempting T1). T3 tries to acquire the lock that T1 holds, but gets stuck waiting, because T1 still holds it. If T2 starts to run, it will have higher priority than T1, and thus it will run. T3, which is higher priority than T2, is stuck waiting for T1, which may never run now that T2 is running. Isn't it sad that the mighty T3 can't run, while lowly T2 controls the CPU? Having high priority just ain't what it used to be.

You can address the priority inversion problem in a number of ways. In the specific case where spin locks cause the problem, you can avoid using spin locks (described more below). More generally, a higher-priority thread waiting for a lower-priority thread can temporarily boost the lower thread's priority, thus enabling it to run and overcoming the inversion, a technique known as **priority inheritance**. A last solution is simplest: ensure all threads have the same priority.

Finally, you might notice the perceived race condition in the solution, just before the call to `park()`. With just the wrong timing, a thread will be about to park, assuming that it should sleep until the lock is no longer held. A switch at that time to another thread (say, a thread holding the lock) could lead to trouble, for example, if that thread then released the lock. The subsequent park by the first thread would then sleep forever (potentially), a problem sometimes called the **wakeup/waiting race**.

Solaris solves this problem by adding a third system call: `setpark()`. By calling this routine, a thread can indicate it is *about to park*. If it then happens to be interrupted and another thread calls `unpark` before `park` is actually called, the subsequent `park` returns immediately instead of sleeping. The code modification, inside of `lock()`, is quite small:

```
1     queue_add(m->q, gettid());
2     setpark(); // new code
3     m->guard = 0;
```

A different solution could pass the guard into the kernel. In that case, the kernel could take precautions to atomically release the lock and de-queue the running thread.

## 28.15 Different OS, Different Support

We have thus far seen one type of support that an OS can provide in order to build a more efficient lock in a thread library. Other OS's provide similar support; the details vary.

For example, Linux provides a **futex** which is similar to the Solaris interface but provides more in-kernel functionality. Specifically, each futex has associated with it a specific physical memory location, as well as a per-futex in-kernel queue. Callers can use futex calls (described below) to sleep and wake as need be.

Specifically, two calls are available. The call to `futex_wait(address, expected)` puts the calling thread to sleep, assuming the value at `address` is equal to `expected`. If it is *not* equal, the call returns immediately. The call to the routine `futex_wake(address)` wakes one thread that is waiting on the queue. The usage of these calls in a Linux mutex is shown in Figure 28.10 (page 19).

This code snippet from `lowlevellock.h` in the nptl library (part of the gnu libc library) [L09] is interesting for a few reasons. First, it uses a single integer to track both whether the lock is held or not (the high bit of the integer) and the number of waiters on the lock (all the other bits). Thus, if the lock is negative, it is held (because the high bit is set and that bit determines the sign of the integer).

Second, the code snippet shows how to optimize for the common case, specifically when there is no contention for the lock; with only one thread acquiring and releasing a lock, very little work is done (the atomic bit test-and-set to lock and an atomic add to release the lock).

See if you can puzzle through the rest of this “real-world” lock to understand how it works. Do it and become a master of Linux locking, or at least somebody who listens when a book tells you to do something<sup>1</sup>.

---

<sup>1</sup>Like buy a print copy of OSTEP! Even though the book is available for free online, wouldn't you just love a hard cover for your desk? Or, better yet, ten copies to share with friends and family? And maybe one extra copy to throw at an enemy? (the book is heavy, and thus chucking it is surprisingly effective)

```

1 void mutex_lock (int *mutex) {
2     int v;
3     /* Bit 31 was clear, we got the mutex (this is the fastpath) */
4     if (atomic_bit_test_set (mutex, 31) == 0)
5         return;
6     atomic_increment (mutex);
7     while (1) {
8         if (atomic_bit_test_set (mutex, 31) == 0) {
9             atomic_decrement (mutex);
10            return;
11        }
12        /* We have to wait now. First make sure the futex value
13           we are monitoring is truly negative (i.e. locked). */
14        v = *mutex;
15        if (v >= 0)
16            continue;
17        futex_wait (mutex, v);
18    }
19 }
20
21 void mutex_unlock (int *mutex) {
22     /* Adding 0x80000000 to the counter results in 0 if and only if
23        there are not other interested threads */
24     if (atomic_add_zero (mutex, 0x80000000))
25         return;
26
27     /* There are other threads waiting for this mutex,
28        wake one of them up. */
29     futex_wake (mutex);
30 }
```

Figure 28.10: Linux-based Futex Locks

## 28.16 Two-Phase Locks

One final note: the Linux approach has the flavor of an old approach that has been used on and off for years, going at least as far back to Dahm Locks in the early 1960's [M82], and is now referred to as a **two-phase lock**. A two-phase lock realizes that spinning can be useful, particularly if the lock is about to be released. So in the first phase, the lock spins for a while, hoping that it can acquire the lock.

However, if the lock is not acquired during the first spin phase, a second phase is entered, where the caller is put to sleep, and only woken up when the lock becomes free later. The Linux lock above is a form of such a lock, but it only spins once; a generalization of this could spin in a loop for a fixed amount of time before using **futex** support to sleep.

Two-phase locks are yet another instance of a **hybrid** approach, where combining two good ideas may indeed yield a better one. Of course, whether it does depends strongly on many things, including the hardware environment, number of threads, and other workload details. As always, making a single general-purpose lock, good for all possible use cases, is quite a challenge.

## 28.17 Summary

The above approach shows how real locks are built these days: some hardware support (in the form of a more powerful instruction) plus some operating system support (e.g., in the form of `park()` and `unpark()` primitives on Solaris, or `futex` on Linux). Of course, the details differ, and the exact code to perform such locking is usually highly tuned. Check out the Solaris or Linux code bases if you want to see more details; they are a fascinating read [L09, S09]. Also see David et al.'s excellent work for a comparison of locking strategies on modern multiprocessors [D+13].

## References

[D91] "Just Win, Baby: Al Davis and His Raiders"

Glenn Dickey, Harcourt 1991

*There is even an undoubtedly bad book about Al Davis and his famous "just win" quote. Or, we suppose, the book is more about Al Davis and the Raiders, and maybe not just the quote. Read the book to find out?*

[D+13] "Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask"

Tudor David, Rachid Guerraoui, Vasileios Trigonakis

SOSP '13, Nemacolin Woodlands Resort, Pennsylvania, November 2013

*An excellent recent paper comparing many different ways to build locks using hardware primitives. A great read to see how many ideas over the years work on modern hardware.*

[D68] "Cooperating sequential processes"

Edsger W. Dijkstra, 1968

Available: <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF>

*One of the early seminal papers in the area. Discusses how Dijkstra posed the original concurrency problem, and Dekker's solution.*

[H93] "MIPS R4000 Microprocessor User's Manual"

Joe Heinrich, Prentice-Hall, June 1993

Available: <http://cag.csail.mit.edu/raw/documents/R4400.Uman.book.Ed2.pdf>

[H91] "Wait-free Synchronization"

Maurice Herlihy

ACM Transactions on Programming Languages and Systems (TOPLAS)

Volume 13, Issue 1, January 1991

*A landmark paper introducing a different approach to building concurrent data structures. However, because of the complexity involved, many of these ideas have been slow to gain acceptance in deployed systems.*

[L81] "Observations on the Development of an Operating System"

Hugh Lauer

SOSP '81, Pacific Grove, California, December 1981

*A must-read retrospective about the development of the Pilot OS, an early PC operating system. Fun and full of insights.*

[L09] "glibc 2.9 (include Linux pthreads implementation)"

Available: <http://ftp.gnu.org/gnu/glibc/>

*In particular, take a look at the nptl subdirectory where you will find most of the pthread support in Linux today.*

[M82] "The Architecture of the Burroughs B5000

20 Years Later and Still Ahead of the Times?"

Alastair J.W. Mayer, 1982

[www.ajwm.net/amayer/papers/B5000.html](http://www.ajwm.net/amayer/papers/B5000.html)

*From the paper: "One particularly useful instruction is the RDLK (read-lock). It is an indivisible operation which reads from and writes into a memory location." RDLK is thus an early test-and-set primitive, if not the earliest. Some credit here goes to an engineer named Dave Dahm, who apparently invented a number of these things for the Burroughs systems, including a form of spin locks (called "Buzz Locks") as well as a two-phase lock eponymously called "Dahm Locks."*

[M15] “OSSpinLock Is Unsafe”

John McCall

Available: [mjtai.com/blog/2015/12/16/osspinlock-is-unsafe](http://mjtai.com/blog/2015/12/16/osspinlock-is-unsafe)

*A short post about why calling OSSpinLock on Mac OS X is unsafe when using threads of different priorities – you might end up spinning forever! So be careful, Mac fanatics, even your mighty system sometimes is less than perfect...*

[MS91] “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors”

John M. Mellor-Crummey and M. L. Scott

ACM TOCS, Volume 9, Issue 1, February 1991

*An excellent and thorough survey on different locking algorithms. However, no operating systems support is used, just fancy hardware instructions.*

[P81] “Myths About the Mutual Exclusion Problem”

G.L. Peterson

Information Processing Letters, 12(3), pages 115–116, 1981

*Peterson’s algorithm introduced here.*

[R97] “What Really Happened on Mars?”

Glenn E. Reeves

Available: [research.microsoft.com/en-us/um/people/mbj/Mars\\_Pathfinder/Authoritative\\_Account.html](http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html)

*A detailed description of priority inversion on the Mars Pathfinder robot. This low-level concurrent code matters a lot, especially in space!*

[S05] “Guide to porting from Solaris to Linux on x86”

Ajay Sood, April 29, 2005

Available: <http://www.ibm.com/developerworks/linux/library/l-solar/>

[S09] “OpenSolaris Thread Library”

Available: <http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/lib/libc/port/threads/synch.c>

*This is also pretty interesting to look at, though who knows what will happen to it now that Oracle owns Sun. Thanks to Mike Swift for the pointer to the code.*

[W09] “Load-Link, Store-Conditional”

Wikipedia entry on said topic, as of October 22, 2009

[http://en.wikipedia.org/wiki/Load-Link\\_Store-Conditional](http://en.wikipedia.org/wiki/Load-Link_Store-Conditional)

*Can you believe we referenced wikipedia? Pretty lazy, no? But, we found the information there first, and it felt wrong not to cite it. Further, they even listed the instructions for the different architectures: `ldl_1/st1_c` and `ldq_1/stq_c` (Alpha), `lwarx/stwcx` (PowerPC), `l1/sc` (MIPS), and `ldrex/strex` (ARM version 6 and above). Actually wikipedia is pretty amazing, so don’t be so harsh, OK?*

[WG00] “The SPARC Architecture Manual: Version 9”

David L. Weaver and Tom Germond, September 2000

SPARC International, San Jose, California

Available: <http://www.sparc.org/standards/SPARCV9.pdf>

*Also see: [http://developers.sun.com/solaris/articles/atomic\\_sparc/](http://developers.sun.com/solaris/articles/atomic_sparc/) for some more details on Sparc atomic operations.*

## Homework

This program, `x86.py`, allows you to see how different thread interleavings either cause or avoid race conditions. See the README for details on how the program works and its basic inputs, then answer the questions below.

### Questions

1. First let's get ready to run `x86.py` with the flag `-p flag.s`. This code "implements" locking with a single memory flag. Can you understand what the assembly code is trying to do?
2. When you run with the defaults, does `flag.s` work as expected? Does it produce the correct result? Use the `-M` and `-R` flags to trace variables and registers (and turn on `-c` to see their values). Can you predict what value will end up in `flag` as the code runs?
3. Change the value of the register `%bx` with the `-a` flag (e.g., `-a bx=2,bx=2` if you are running just two threads). What does the code do? How does it change your answer for the question above?
4. Set `bx` to a high value for each thread, and then use the `-i` flag to generate different interrupt frequencies; what values lead to a bad outcomes? Which lead to good outcomes?
5. Now let's look at the program `test-and-set.s`. First, try to understand the code, which uses the `xchg` instruction to build a simple locking primitive. How is the lock acquire written? How about lock release?
6. Now run the code, changing the value of the interrupt interval (`-i`) again, and making sure to loop for a number of times. Does the code always work as expected? Does it sometimes lead to an inefficient use of the CPU? How could you quantify that?
7. Use the `-P` flag to generate specific tests of the locking code. For example, run a schedule that grabs the lock in the first thread, but then tries to acquire it in the second. Does the right thing happen? What else should you test?
8. Now let's look at the code in `peterson.s`, which implements Peterson's algorithm (mentioned in a sidebar in the text). Study the code and see if you can make sense of it.
9. Now run the code with different values of `-i`. What kinds of different behavior do you see?
10. Can you control the scheduling (with the `-P` flag) to "prove" that the code works? What are the different cases you should show hold? Think about mutual exclusion and deadlock avoidance.

11. Now study the code for the ticket lock in `ticket.s`. Does it match the code in the chapter?
12. Now run the code, with the following flags: `-a bx=1000, bx=1000` (this flag sets each thread to loop through the critical 1000 times). Watch what happens over time; do the threads spend much time spinning waiting for the lock?
13. How does the code behave as you add more threads?
14. Now examine `yield.s`, in which we pretend that a `yield` instruction enables one thread to yield control of the CPU to another (realistically, this would be an OS primitive, but for the simplicity of simulation, we assume there is an instruction that does the task). Find a scenario where `test-and-set.s` wastes cycles spinning, but `yield.s` does not. How many instructions are saved? In what scenarios do these savings arise?
15. Finally, examine `test-and-test-and-set.s`. What does this lock do? What kind of savings does it introduce as compared to `test-and-set.s`?

## Lock-based Concurrent Data Structures

Before moving beyond locks, we'll first describe how to use locks in some common data structures. Adding locks to a data structure to make it usable by threads makes the structure **thread safe**. Of course, exactly how such locks are added determines both the correctness and performance of the data structure. And thus, our challenge:

### CRUX: HOW TO ADD LOCKS TO DATA STRUCTURES

When given a particular data structure, how should we add locks to it, in order to make it work correctly? Further, how do we add locks such that the data structure yields high performance, enabling many threads to access the structure at once, i.e., **concurrently**?

Of course, we will be hard pressed to cover all data structures or all methods for adding concurrency, as this is a topic that has been studied for years, with (literally) thousands of research papers published about it. Thus, we hope to provide a sufficient introduction to the type of thinking required, and refer you to some good sources of material for further inquiry on your own. We found Moir and Shavit's survey to be a great source of information [MS04].

### 29.1 Concurrent Counters

One of the simplest data structures is a counter. It is a structure that is commonly used and has a simple interface. We define a simple non-concurrent counter in Figure 29.1.

#### Simple But Not Scalable

As you can see, the non-synchronized counter is a trivial data structure, requiring a tiny amount of code to implement. We now have our next challenge: how can we make this code **thread safe**? Figure 29.2 shows how we do so.

```

1  typedef struct __counter_t {
2      int value;
3  } counter_t;
4
5  void init(counter_t *c) {
6      c->value = 0;
7  }
8
9  void increment(counter_t *c) {
10     c->value++;
11 }
12
13 void decrement(counter_t *c) {
14     c->value--;
15 }
16
17 int get(counter_t *c) {
18     return c->value;
19 }
```

Figure 29.1: A Counter Without Locks

```

1  typedef struct __counter_t {
2      int value;
3      pthread_mutex_t lock;
4  } counter_t;
5
6  void init(counter_t *c) {
7      c->value = 0;
8      Pthread_mutex_init(&c->lock, NULL);
9  }
10
11 void increment(counter_t *c) {
12     Pthread_mutex_lock(&c->lock);
13     c->value++;
14     Pthread_mutex_unlock(&c->lock);
15 }
16
17 void decrement(counter_t *c) {
18     Pthread_mutex_lock(&c->lock);
19     c->value--;
20     Pthread_mutex_unlock(&c->lock);
21 }
22
23 int get(counter_t *c) {
24     Pthread_mutex_lock(&c->lock);
25     int rc = c->value;
26     Pthread_mutex_unlock(&c->lock);
27     return rc;
28 }
```

Figure 29.2: A Counter With Locks

This concurrent counter is simple and works correctly. In fact, it follows a design pattern common to the simplest and most basic concurrent data structures: it simply adds a single lock, which is acquired when calling a routine that manipulates the data structure, and is released when returning from the call. In this manner, it is similar to a data structure built with **monitors** [BH73], where locks are acquired and released automatically as you call and return from object methods.

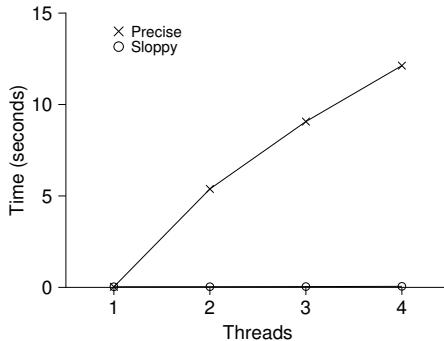


Figure 29.3: Performance of Traditional vs. Sloppy Counters

At this point, you have a working concurrent data structure. The problem you might have is performance. If your data structure is too slow, you'll have to do more than just add a single lock; such optimizations, if needed, are thus the topic of the rest of the chapter. Note that if the data structure is *not* too slow, you are done! No need to do something fancy if something simple will work.

To understand the performance costs of the simple approach, we run a benchmark in which each thread updates a single shared counter a fixed number of times; we then vary the number of threads. Figure 29.3 shows the total time taken, with one to four threads active; each thread updates the counter one million times. This experiment was run upon an iMac with four Intel 2.7 GHz i5 CPUs; with more CPUs active, we hope to get more total work done per unit time.

From the top line in the figure (labeled *precise*), you can see that the performance of the synchronized counter scales poorly. Whereas a single thread can complete the million counter updates in a tiny amount of time (roughly 0.03 seconds), having two threads each update the counter one million times concurrently leads to a massive slowdown (taking over 5 seconds!). It only gets worse with more threads.

Ideally, you'd like to see the threads complete just as quickly on multiple processors as the single thread does on one. Achieving this end is called **perfect scaling**; even though more work is done, it is done in parallel, and hence the time taken to complete the task is not increased.

## Scalable Counting

Amazingly, researchers have studied how to build more scalable counters for years [MS04]. Even more amazing is the fact that scalable counters matter, as recent work in operating system performance analysis has shown [B+10]; without scalable counting, some workloads running on

Time	$L_1$	$L_2$	$L_3$	$L_4$	$G$
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	5 → 0	1	3	4	5 (from $L_1$ )
7	0	2	4	5 → 0	10 (from $L_4$ )

Figure 29.4: Tracing the Sloppy Counters

Linux suffer from serious scalability problems on multicore machines.

Though many techniques have been developed to attack this problem, we'll now describe one particular approach. The idea, introduced in recent research [B+10], is known as a **sloppy counter**.

The sloppy counter works by representing a single logical counter via numerous *local* physical counters, one per CPU core, as well as a single *global* counter. Specifically, on a machine with four CPUs, there are four local counters and one global one. In addition to these counters, there are also locks: one for each local counter, and one for the global counter.

The basic idea of sloppy counting is as follows. When a thread running on a given core wishes to increment the counter, it increments its local counter; access to this local counter is synchronized via the corresponding local lock. Because each CPU has its own local counter, threads across CPUs can update local counters without contention, and thus counter updates are scalable.

However, to keep the global counter up to date (in case a thread wishes to read its value), the local values are periodically transferred to the global counter, by acquiring the global lock and incrementing it by the local counter's value; the local counter is then reset to zero.

How often this local-to-global transfer occurs is determined by a threshold, which we call  $S$  here (for sloppiness). The smaller  $S$  is, the more the counter behaves like the non-scalable counter above; the bigger  $S$  is, the more scalable the counter, but the further off the global value might be from the actual count. One could simply acquire all the local locks and the global lock (in a specified order, to avoid deadlock) to get an exact value, but that is not scalable.

To make this clear, let's look at an example (Figure 29.4). In this example, the threshold  $S$  is set to 5, and there are threads on each of four CPUs updating their local counters  $L_1 \dots L_4$ . The global counter value ( $G$ ) is also shown in the trace, with time increasing downward. At each time step, a local counter may be incremented; if the local value reaches the threshold  $S$ , the local value is transferred to the global counter and the local counter is reset.

The lower line in Figure 29.3 (labeled *sloppy*, on page 3) shows the performance of sloppy counters with a threshold  $S$  of 1024. Performance is excellent; the time taken to update the counter four million times on four processors is hardly higher than the time taken to update it one million times on one processor.

```

1  typedef struct __counter_t {
2      int           global;          // global count
3      pthread_mutex_t glock;        // global lock
4      int           local[NUMCPUS]; // local count (per cpu)
5      pthread_mutex_t llock[NUMCPUS]; // ... and locks
6      int           threshold;     // update frequency
7  } counter_t;
8
9  // init: record threshold, init locks, init values
10 //      of all local counts and global count
11 void init(counter_t *c, int threshold) {
12     c->threshold = threshold;
13     c->global = 0;
14     pthread_mutex_init(&c->glock, NULL);
15     int i;
16     for (i = 0; i < NUMCPUS; i++) {
17         c->local[i] = 0;
18         pthread_mutex_init(&c->llock[i], NULL);
19     }
20 }
21
22 // update: usually, just grab local lock and update local amount
23 //      once local count has risen by 'threshold', grab global
24 //      lock and transfer local values to it
25 void update(counter_t *c, int threadID, int amt) {
26     int cpu = threadID % NUMCPUS;
27     pthread_mutex_lock(&c->llock[cpu]);
28     c->local[cpu] += amt;           // assumes amt > 0
29     if (c->local[cpu] >= c->threshold) { // transfer to global
30         pthread_mutex_lock(&c->glock);
31         c->global += c->local[cpu];
32         pthread_mutex_unlock(&c->glock);
33         c->local[cpu] = 0;
34     }
35     pthread_mutex_unlock(&c->llock[cpu]);
36 }
37
38 // get: just return global amount (which may not be perfect)
39 int get(counter_t *c) {
40     pthread_mutex_lock(&c->glock);
41     int val = c->global;
42     pthread_mutex_unlock(&c->glock);
43     return val; // only approximate!
44 }
```

Figure 29.5: Sloppy Counter Implementation

Figure 29.6 shows the importance of the threshold value  $S$ , with four threads each incrementing the counter 1 million times on four CPUs. If  $S$  is low, performance is poor (but the global count is always quite accurate); if  $S$  is high, performance is excellent, but the global count lags (by at most the number of CPUs multiplied by  $S$ ). This accuracy/performance trade-off is what sloppy counters enables.

A rough version of such a sloppy counter is found in Figure 29.5. Read it, or better yet, run it yourself in some experiments to better understand how it works.

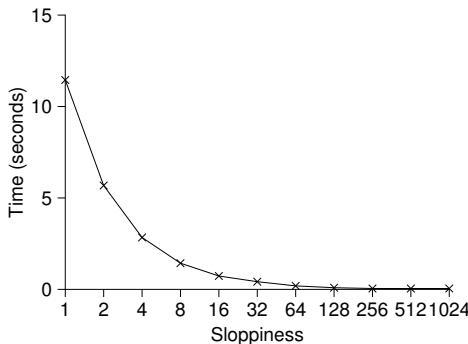


Figure 29.6: Scaling Sloppy Counters

## 29.2 Concurrent Linked Lists

We next examine a more complicated structure, the linked list. Let's start with a basic approach once again. For simplicity, we'll omit some of the obvious routines that such a list would have and just focus on concurrent insert; we'll leave it to the reader to think about lookup, delete, and so forth. Figure 29.7 shows the code for this rudimentary data structure.

As you can see in the code, the code simply acquires a lock in the insert routine upon entry, and releases it upon exit. One small tricky issue arises if `malloc()` happens to fail (a rare case); in this case, the code must also release the lock before failing the insert.

This kind of exceptional control flow has been shown to be quite error prone; a recent study of Linux kernel patches found that a huge fraction of bugs (nearly 40%) are found on such rarely-taken code paths (indeed, this observation sparked some of our own research, in which we removed all memory-failing paths from a Linux file system, resulting in a more robust system [S+11]).

Thus, a challenge: can we rewrite the insert and lookup routines to remain correct under concurrent insert but avoid the case where the failure path also requires us to add the call to unlock?

The answer, in this case, is yes. Specifically, we can rearrange the code a bit so that the lock and release only surround the actual critical section in the insert code, and that a common exit path is used in the lookup code. The former works because part of the lookup actually need not be locked; assuming that `malloc()` itself is thread-safe, each thread can call into it without worry of race conditions or other concurrency bugs. Only when updating the shared list does a lock need to be held. See Figure 29.8 for the details of these modifications.

```

1 // basic node structure
2 typedef struct __node_t {
3     int                         key;
4     struct __node_t             *next;
5 } node_t;
6
7 // basic list structure (one used per list)
8 typedef struct __list_t {
9     node_t                      *head;
10    pthread_mutex_t              lock;
11 } list_t;
12
13 void List_Init(list_t *L) {
14     L->head = NULL;
15     pthread_mutex_init(&L->lock, NULL);
16 }
17
18 int List_Insert(list_t *L, int key) {
19     pthread_mutex_lock(&L->lock);
20     node_t *new = malloc(sizeof(node_t));
21     if (new == NULL) {
22         perror("malloc");
23         pthread_mutex_unlock(&L->lock);
24         return -1; // fail
25     }
26     new->key   = key;
27     new->next  = L->head;
28     L->head   = new;
29     pthread_mutex_unlock(&L->lock);
30     return 0; // success
31 }
32
33 int List_Lookup(list_t *L, int key) {
34     pthread_mutex_lock(&L->lock);
35     node_t *curr = L->head;
36     while (curr) {
37         if (curr->key == key) {
38             pthread_mutex_unlock(&L->lock);
39             return 0; // success
40         }
41         curr = curr->next;
42     }
43     pthread_mutex_unlock(&L->lock);
44     return -1; // failure
45 }
```

Figure 29.7: Concurrent Linked List

As for the lookup routine, it is a simple code transformation to jump out of the main search loop to a single return path. Doing so again reduces the number of lock acquire/release points in the code, and thus decreases the chances of accidentally introducing bugs (such as forgetting to unlock before returning) into the code.

### Scaling Linked Lists

Though we again have a basic concurrent linked list, once again we are in a situation where it does not scale particularly well. One technique that researchers have explored to enable more concurrency within a list is

```

1 void List_Init(list_t *L) {
2     L->head = NULL;
3     pthread_mutex_init(&L->lock, NULL);
4 }
5
6 void List_Insert(list_t *L, int key) {
7     // synchronization not needed
8     node_t *new = malloc(sizeof(node_t));
9     if (new == NULL) {
10         perror("malloc");
11         return;
12     }
13     new->key = key;
14
15     // just lock critical section
16     pthread_mutex_lock(&L->lock);
17     new->next = L->head;
18     L->head = new;
19     pthread_mutex_unlock(&L->lock);
20 }
21
22 int List_Lookup(list_t *L, int key) {
23     int rv = -1;
24     pthread_mutex_lock(&L->lock);
25     node_t *curr = L->head;
26     while (curr) {
27         if (curr->key == key) {
28             rv = 0;
29             break;
30         }
31         curr = curr->next;
32     }
33     pthread_mutex_unlock(&L->lock);
34     return rv; // now both success and failure
35 }
```

Figure 29.8: Concurrent Linked List: Rewritten

something called **hand-over-hand locking** (a.k.a. **lock coupling**) [MS04].

The idea is pretty simple. Instead of having a single lock for the entire list, you instead add a lock per node of the list. When traversing the list, the code first grabs the next node's lock and then releases the current node's lock (which inspires the name hand-over-hand).

Conceptually, a hand-over-hand linked list makes some sense; it enables a high degree of concurrency in list operations. However, in practice, it is hard to make such a structure faster than the simple single lock approach, as the overheads of acquiring and releasing locks for each node of a list traversal is prohibitive. Even with very large lists, and a large number of threads, the concurrency enabled by allowing multiple ongoing traversals is unlikely to be faster than simply grabbing a single lock, performing an operation, and releasing it. Perhaps some kind of hybrid (where you grab a new lock every so many nodes) would be worth investigating.

**TIP: MORE CONCURRENCY ISN'T NECESSARILY FASTER**

If the scheme you design adds a lot of overhead (for example, by acquiring and releasing locks frequently, instead of once), the fact that it is more concurrent may not be important. Simple schemes tend to work well, especially if they use costly routines rarely. Adding more locks and complexity can be your downfall. All of that said, there is one way to really know: build both alternatives (simple but less concurrent, and complex but more concurrent) and measure how they do. In the end, you can't cheat on performance; your idea is either faster, or it isn't.

**TIP: BE WARY OF LOCKS AND CONTROL FLOW**

A general design tip, which is useful in concurrent code as well as elsewhere, is to be wary of control flow changes that lead to function returns, exits, or other similar error conditions that halt the execution of a function. Because many functions will begin by acquiring a lock, allocating some memory, or doing other similar stateful operations, when errors arise, the code has to undo all of the state before returning, which is error-prone. Thus, it is best to structure code to minimize this pattern.

### 29.3 Concurrent Queues

As you know by now, there is always a standard method to make a concurrent data structure: add a big lock. For a queue, we'll skip that approach, assuming you can figure it out.

Instead, we'll take a look at a slightly more concurrent queue designed by Michael and Scott [MS98]. The data structures and code used for this queue are found in Figure 29.9 on the following page.

If you study this code carefully, you'll notice that there are two locks, one for the head of the queue, and one for the tail. The goal of these two locks is to enable concurrency of enqueue and dequeue operations. In the common case, the enqueue routine will only access the tail lock, and dequeue only the head lock.

One trick used by Michael and Scott is to add a dummy node (allocated in the queue initialization code); this dummy enables the separation of head and tail operations. Study the code, or better yet, type it in, run it, and measure it, to understand how it works deeply.

Queues are commonly used in multi-threaded applications. However, the type of queue used here (with just locks) often does not completely meet the needs of such programs. A more fully developed bounded queue, that enables a thread to wait if the queue is either empty or overly full, is the subject of our intense study in the next chapter on condition variables. Watch for it!

```

1  typedef struct __node_t {
2      int             value;
3      struct __node_t *next;
4  } node_t;
5
6  typedef struct __queue_t {
7      node_t          *head;
8      node_t          *tail;
9      pthread_mutex_t headLock;
10     pthread_mutex_t tailLock;
11 } queue_t;
12
13 void Queue_Init(queue_t *q) {
14     node_t *tmp = malloc(sizeof(node_t));
15     tmp->next = NULL;
16     q->head = q->tail = tmp;
17     pthread_mutex_init(&q->headLock, NULL);
18     pthread_mutex_init(&q->tailLock, NULL);
19 }
20
21 void Queue_Enqueue(queue_t *q, int value) {
22     node_t *tmp = malloc(sizeof(node_t));
23     assert(tmp != NULL);
24     tmp->value = value;
25     tmp->next = NULL;
26
27     pthread_mutex_lock(&q->tailLock);
28     q->tail->next = tmp;
29     q->tail = tmp;
30     pthread_mutex_unlock(&q->tailLock);
31 }
32
33 int Queue_Dequeue(queue_t *q, int *value) {
34     pthread_mutex_lock(&q->headLock);
35     node_t *tmp = q->head;
36     node_t *newHead = tmp->next;
37     if (newHead == NULL) {
38         pthread_mutex_unlock(&q->headLock);
39         return -1; // queue was empty
40     }
41     *value = newHead->value;
42     q->head = newHead;
43     pthread_mutex_unlock(&q->headLock);
44     free(tmp);
45     return 0;
46 }

```

Figure 29.9: Michael and Scott Concurrent Queue

## 29.4 Concurrent Hash Table

We end our discussion with a simple and widely applicable concurrent data structure, the hash table. We'll focus on a simple hash table that does not resize; a little more work is required to handle resizing, which we leave as an exercise for the reader (sorry!).

This concurrent hash table is straightforward, is built using the concurrent lists we developed earlier, and works incredibly well. The reason

```

1 #define BUCKETS (101)
2
3 typedef struct __hash_t {
4     list_t lists[BUCKETS];
5 } hash_t;
6
7 void Hash_Init(hash_t *H) {
8     int i;
9     for (i = 0; i < BUCKETS; i++) {
10         List_Init(&H->lists[i]);
11     }
12 }
13
14 int Hash_Insert(hash_t *H, int key) {
15     int bucket = key % BUCKETS;
16     return List_Insert(&H->lists[bucket], key);
17 }
18
19 int Hash_Lookup(hash_t *H, int key) {
20     int bucket = key % BUCKETS;
21     return List_Lookup(&H->lists[bucket], key);
22 }
```

Figure 29.10: A Concurrent Hash Table

for its good performance is that instead of having a single lock for the entire structure, it uses a lock per hash bucket (each of which is represented by a list). Doing so enables many concurrent operations to take place.

Figure 29.11 shows the performance of the hash table under concurrent updates (from 10,000 to 50,000 concurrent updates from each of four threads, on the same iMac with four CPUs). Also shown, for the sake of comparison, is the performance of a linked list (with a single lock). As you can see from the graph, this simple concurrent hash table scales magnificently; the linked list, in contrast, does not.

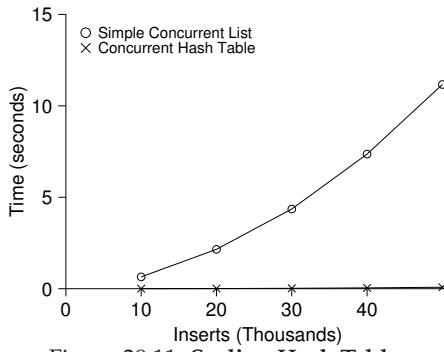


Figure 29.11: Scaling Hash Tables

**TIP: AVOID PREMATURE OPTIMIZATION (KNUTH'S LAW)**

When building a concurrent data structure, start with the most basic approach, which is to add a single big lock to provide synchronized access. By doing so, you are likely to build a *correct* lock; if you then find that it suffers from performance problems, you can refine it, thus only making it fast if need be. As **Knuth** famously stated, "Premature optimization is the root of all evil."

Many operating systems utilized a single lock when first transitioning to multiprocessors, including Sun OS and Linux. In the latter, this lock even had a name, the **big kernel lock (BKL)**. For many years, this simple approach was a good one, but when multi-CPU systems became the norm, only allowing a single active thread in the kernel at a time became a performance bottleneck. Thus, it was finally time to add the optimization of improved concurrency to these systems. Within Linux, the more straightforward approach was taken: replace one lock with many. Within Sun, a more radical decision was made: build a brand new operating system, known as Solaris, that incorporates concurrency more fundamentally from day one. Read the Linux and Solaris kernel books for more information about these fascinating systems [BC05, MM00].

## 29.5 Summary

We have introduced a sampling of concurrent data structures, from counters, to lists and queues, and finally to the ubiquitous and heavily-used hash table. We have learned a few important lessons along the way: to be careful with acquisition and release of locks around control flow changes; that enabling more concurrency does not necessarily increase performance; that performance problems should only be remedied once they exist. This last point, of avoiding **premature optimization**, is central to any performance-minded developer; there is no value in making something faster if doing so will not improve the overall performance of the application.

Of course, we have just scratched the surface of high performance structures. See Moir and Shavit's excellent survey for more information, as well as links to other sources [MS04]. In particular, you might be interested in other structures (such as B-trees); for this knowledge, a database class is your best bet. You also might be interested in techniques that don't use traditional locks at all; such **non-blocking data structures** are something we'll get a taste of in the chapter on common concurrency bugs, but frankly this topic is an entire area of knowledge requiring more study than is possible in this humble book. Find out more on your own if you are interested (as always!).

## References

- [B+10] "An Analysis of Linux Scalability to Many Cores"  
 Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek,  
 Robert Morris, Nickolai Zeldovich  
 OSDI '10, Vancouver, Canada, October 2010  
*A great study of how Linux performs on multicore machines, as well as some simple solutions.*
- [BH73] "Operating System Principles"  
 Per Brinch Hansen, Prentice-Hall, 1973  
 Available: <http://portal.acm.org/citation.cfm?id=540365>  
*One of the first books on operating systems; certainly ahead of its time. Introduced monitors as a concurrency primitive.*
- [BC05] "Understanding the Linux Kernel (Third Edition)"  
 Daniel P. Bovet and Marco Cesati  
 O'Reilly Media, November 2005  
*The classic book on the Linux kernel. You should read it.*
- [L+13] "A Study of Linux File System Evolution"  
 Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Shan Lu  
 FAST '13, San Jose, CA, February 2013  
*Our paper that studies every patch to Linux file systems over nearly a decade. Lots of fun findings in there; read it to see! The work was painful to do though; the poor graduate student, Lanyue Lu, had to look through every single patch by hand in order to understand what they did.*
- [MS98] "Nonblocking Algorithms and Preemption-safe Locking on Multiprogrammed Shared-memory Multiprocessors"  
 M. Michael and M. Scott  
 Journal of Parallel and Distributed Computing, Vol. 51, No. 1, 1998  
*Professor Scott and his students have been at the forefront of concurrent algorithms and data structures for many years; check out his web page, numerous papers, or books to find out more.*
- [MS04] "Concurrent Data Structures"  
 Mark Moir and Nir Shavit  
 In Handbook of Data Structures and Applications  
 (Editors D. Metha and S.Sahni)  
 Chapman and Hall/CRC Press, 2004  
 Available: [www.cs.tau.ac.il/~shanir/concurrent-data-structures.pdf](http://www.cs.tau.ac.il/~shanir/concurrent-data-structures.pdf)  
*A short but relatively comprehensive reference on concurrent data structures. Though it is missing some of the latest works in the area (due to its age), it remains an incredibly useful reference.*
- [MM00] "Solaris Internals: Core Kernel Architecture"  
 Jim Mauro and Richard McDougall  
 Prentice Hall, October 2000  
*The Solaris book. You should also read this, if you want to learn in great detail about something other than Linux.*
- [S+11] "Making the Common Case the Only Case with Anticipatory Memory Allocation"  
 Swaminathan Sundararaman, Yupu Zhang, Sriram Subramanian,  
 Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau  
 FAST '11, San Jose, CA, February 2011  
*Our work on removing possibly-failing calls to malloc from kernel code paths. The idea is to allocate all potentially needed memory before doing any of the work, thus avoiding failure deep down in the storage stack.*

## Condition Variables

Thus far we have developed the notion of a lock and seen how one can be properly built with the right combination of hardware and OS support. Unfortunately, locks are not the only primitives that are needed to build concurrent programs.

In particular, there are many cases where a thread wishes to check whether a **condition** is true before continuing its execution. For example, a parent thread might wish to check whether a child thread has completed before continuing (this is often called a `join()`); how should such a wait be implemented? Let's look at Figure 30.1.

```

1 void *child(void *arg) {
2     printf("child\n");
3     // XXX how to indicate we are done?
4     return NULL;
5 }
6
7 int main(int argc, char *argv[]) {
8     printf("parent: begin\n");
9     pthread_t c;
10    Pthread_create(&c, NULL, child, NULL); // create child
11    // XXX how to wait for child?
12    printf("parent: end\n");
13    return 0;
14 }
```

Figure 30.1: A Parent Waiting For Its Child

What we would like to see here is the following output:

```
parent: begin
child
parent: end
```

We could try using a shared variable, as you see in Figure 30.2. This solution will generally work, but it is hugely inefficient as the parent spins and wastes CPU time. What we would like here instead is some way to put the parent to sleep until the condition we are waiting for (e.g., the child is done executing) comes true.

```

1  volatile int done = 0;
2
3  void *child(void *arg) {
4      printf("child\n");
5      done = 1;
6      return NULL;
7  }
8
9  int main(int argc, char *argv[]) {
10     printf("parent: begin\n");
11     pthread_t c;
12     Pthread_create(&c, NULL, child, NULL); // create child
13     while (done == 0)
14         ; // spin
15     printf("parent: end\n");
16     return 0;
17 }
```

Figure 30.2: Parent Waiting For Child: Spin-based Approach

#### THE CRUX: HOW TO WAIT FOR A CONDITION

In multi-threaded programs, it is often useful for a thread to wait for some condition to become true before proceeding. The simple approach, of just spinning until the condition becomes true, is grossly inefficient and wastes CPU cycles, and in some cases, can be incorrect. Thus, how should a thread wait for a condition?

### 30.1 Definition and Routines

To wait for a condition to become true, a thread can make use of what is known as a **condition variable**. A **condition variable** is an explicit queue that threads can put themselves on when some state of execution (i.e., some **condition**) is not as desired (by **waiting** on the condition); some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue (by **signaling** on the condition). The idea goes back to Dijkstra's use of "private semaphores" [D68]; a similar idea was later named a "condition variable" by Hoare in his work on monitors [H74].

To declare such a condition variable, one simply writes something like this: `pthread_cond_t c;`, which declares `c` as a condition variable (note: proper initialization is also required). A condition variable has two operations associated with it: `wait()` and `signal()`. The `wait()` call is executed when a thread wishes to put itself to sleep; the `signal()` call is executed when a thread has changed something in the program and thus wants to wake a sleeping thread waiting on this condition. Specifically, the POSIX calls look like this:

```

pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

```

1 int done = 0;
2 pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3 pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5 void thr_exit() {
6     Pthread_mutex_lock(&m);
7     done = 1;
8     Pthread_cond_signal(&c);
9     Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

**Figure 30.3: Parent Waiting For Child: Use A Condition Variable**

We will often refer to these as `wait()` and `signal()` for simplicity. One thing you might notice about the `wait()` call is that it also takes a mutex as a parameter; it assumes that this mutex is locked when `wait()` is called. The responsibility of `wait()` is to release the lock and put the calling thread to sleep (atomically); when the thread wakes up (after some other thread has signaled it), it must re-acquire the lock before returning to the caller. This complexity stems from the desire to prevent certain race conditions from occurring when a thread is trying to put itself to sleep. Let's take a look at the solution to the join problem (Figure 30.3) to understand this better.

There are two cases to consider. In the first, the parent creates the child thread but continues running itself (assume we have only a single processor) and thus immediately calls into `thr_join()` to wait for the child thread to complete. In this case, it will acquire the lock, check if the child is done (it is not), and put itself to sleep by calling `wait()` (hence releasing the lock). The child will eventually run, print the message "child", and call `thr_exit()` to wake the parent thread; this code just grabs the lock, sets the state variable `done`, and signals the parent thus waking it. Finally, the parent will run (returning from `wait()` with the lock held), unlock the lock, and print the final message "parent: end".

In the second case, the child runs immediately upon creation, sets `done` to 1, calls `signal` to wake a sleeping thread (but there is none, so it just returns), and is done. The parent then runs, calls `thr_join()`, sees that `done` is 1, and thus does not wait and returns.

One last note: you might observe the parent uses a `while` loop instead of just an `if` statement when deciding whether to wait on the condition. While this does not seem strictly necessary per the logic of the program, it is always a good idea, as we will see below.

To make sure you understand the importance of each piece of the `thr_exit()` and `thr_join()` code, let's try a few alternate implementations. First, you might be wondering if we need the state variable `done`. What if the code looked like the example below? Would this work?

```

1 void thr_exit() {
2     Pthread_mutex_lock(&m);
3     Pthread_cond_signal(&c);
4     Pthread_mutex_unlock(&m);
5 }
6
7 void thr_join() {
8     Pthread_mutex_lock(&m);
9     Pthread_cond_wait(&c, &m);
10    Pthread_mutex_unlock(&m);
11 }
```

Unfortunately this approach is broken. Imagine the case where the child runs immediately and calls `thr_exit()` immediately; in this case, the child will signal, but there is no thread asleep on the condition. When the parent runs, it will simply call `wait` and be stuck; no thread will ever wake it. From this example, you should appreciate the importance of the state variable `done`; it records the value the threads are interested in knowing. The sleeping, waking, and locking all are built around it.

Here is another poor implementation. In this example, we imagine that one does not need to hold a lock in order to signal and wait. What problem could occur here? Think about it!

```

1 void thr_exit() {
2     done = 1;
3     Pthread_cond_signal(&c);
4 }
5
6 void thr_join() {
7     if (done == 0)
8         Pthread_cond_wait(&c);
9 }
```

The issue here is a subtle race condition. Specifically, if the parent calls `thr_join()` and then checks the value of `done`, it will see that it is 0 and thus try to go to sleep. But just before it calls `wait` to go to sleep, the parent is interrupted, and the child runs. The child changes the state variable `done` to 1 and signals, but no thread is waiting and thus no thread is woken. When the parent runs again, it sleeps forever, which is sad.

**TIP: ALWAYS HOLD THE LOCK WHILE SIGNALING**

Although it is strictly not necessary in all cases, it is likely simplest and best to hold the lock while signaling when using condition variables. The example above shows a case where you *must* hold the lock for correctness; however, there are some other cases where it is likely OK not to, but probably is something you should avoid. Thus, for simplicity, **hold the lock when calling signal**.

The converse of this tip, i.e., hold the lock when calling wait, is not just a tip, but rather mandated by the semantics of wait, because wait always (a) assumes the lock is held when you call it, (b) releases said lock when putting the caller to sleep, and (c) re-acquires the lock just before returning. Thus, the generalization of this tip is correct: **hold the lock when calling signal or wait**, and you will always be in good shape.

Hopefully, from this simple join example, you can see some of the basic requirements of using condition variables properly. To make sure you understand, we now go through a more complicated example: the **producer/consumer** or **bounded-buffer** problem.

## 30.2 The Producer/Consumer (Bounded Buffer) Problem

The next synchronization problem we will confront in this chapter is known as the **producer/consumer** problem, or sometimes as the **bounded buffer** problem, which was first posed by Dijkstra [D72]. Indeed, it was this very producer/consumer problem that led Dijkstra and his co-workers to invent the generalized semaphore (which can be used as either a lock or a condition variable) [D01]; we will learn more about semaphores later.

Imagine one or more producer threads and one or more consumer threads. Producers generate data items and place them in a buffer; consumers grab said items from the buffer and consume them in some way.

This arrangement occurs in many real systems. For example, in a multi-threaded web server, a producer puts HTTP requests into a work queue (i.e., the bounded buffer); consumer threads take requests out of this queue and process them.

A bounded buffer is also used when you pipe the output of one program into another, e.g., `grep foo file.txt | wc -l`. This example runs two processes concurrently; `grep` writes lines from `file.txt` with the string `foo` in them to what it thinks is standard output; the UNIX shell redirects the output to what is called a UNIX pipe (created by the `pipe` system call). The other end of this pipe is connected to the standard input of the process `wc`, which simply counts the number of lines in the input stream and prints out the result. Thus, the `grep` process is the producer; the `wc` process is the consumer; between them is an in-kernel bounded buffer; you, in this example, are just the happy user.

```

1 int buffer;
2 int count = 0; // initially, empty
3
4 void put(int value) {
5     assert(count == 0);
6     count = 1;
7     buffer = value;
8 }
9
10 int get() {
11     assert(count == 1);
12     count = 0;
13     return buffer;
14 }
```

Figure 30.4: The Put And Get Routines (Version 1)

```

1 void *producer(void *arg) {
2     int i;
3     int loops = (int) arg;
4     for (i = 0; i < loops; i++) {
5         put(i);
6     }
7 }
8
9 void *consumer(void *arg) {
10    int i;
11    while (1) {
12        int tmp = get();
13        printf("%d\n", tmp);
14    }
15 }
```

Figure 30.5: Producer/Consumer Threads (Version 1)

Because the bounded buffer is a shared resource, we must of course require synchronized access to it, lest<sup>1</sup> a race condition arise. To begin to understand this problem better, let us examine some actual code.

The first thing we need is a shared buffer, into which a producer puts data, and out of which a consumer takes data. Let's just use a single integer for simplicity (you can certainly imagine placing a pointer to a data structure into this slot instead), and the two inner routines to put a value into the shared buffer, and to get a value out of the buffer. See Figure 30.4 for details.

Pretty simple, no? The `put()` routine assumes the buffer is empty (and checks this with an assertion), and then simply puts a value into the shared buffer and marks it full by setting `count` to 1. The `get()` routine does the opposite, setting the buffer to empty (i.e., setting `count` to 0) and returning the value. Don't worry that this shared buffer has just a single entry; later, we'll generalize it to a queue that can hold multiple entries, which will be even more fun than it sounds.

Now we need to write some routines that know when it is OK to access the buffer to either put data into it or get data out of it. The conditions for

---

<sup>1</sup>This is where we drop some serious Old English on you, and the subjunctive form.

```

1 cond_t cond;
2 mutex_t mutex;
3
4 void *producer(void *arg) {
5     int i;
6     for (i = 0; i < loops; i++) {
7         Pthread_mutex_lock(&mutex);           // p1
8         if (count == 1)                      // p2
9             Pthread_cond_wait(&cond, &mutex); // p3
10        put(i);                          // p4
11        Pthread_cond_signal(&cond);       // p5
12        Pthread_mutex_unlock(&mutex);     // p6
13    }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);           // c1
20         if (count == 0)                      // c2
21             Pthread_cond_wait(&cond, &mutex); // c3
22         int tmp = get();                  // c4
23         Pthread_cond_signal(&cond);       // c5
24         Pthread_mutex_unlock(&mutex);     // c6
25         printf("%d\n", tmp);
26     }
27 }
```

Figure 30.6: Producer/Consumer: Single CV And If Statement

this should be obvious: only put data into the buffer when `count` is zero (i.e., when the buffer is empty), and only get data from the buffer when `count` is one (i.e., when the buffer is full). If we write the synchronization code such that a producer puts data into a full buffer, or a consumer gets data from an empty one, we have done something wrong (and in this code, an assertion will fire).

This work is going to be done by two types of threads, one set of which we'll call the **producer** threads, and the other set which we'll call **consumer** threads. Figure 30.5 shows the code for a producer that puts an integer into the shared buffer `loops` number of times, and a consumer that gets the data out of that shared buffer (forever), each time printing out the data item it pulled from the shared buffer.

## A Broken Solution

Now imagine that we have just a single producer and a single consumer. Obviously the `put()` and `get()` routines have critical sections within them, as `put()` updates the buffer, and `get()` reads from it. However, putting a lock around the code doesn't work; we need something more. Not surprisingly, that something more is some condition variables. In this (broken) first try (Figure 30.6), we have a single condition variable `cond` and associated lock `mutex`.

$T_{c1}$	State	$T_{c2}$	State	$T_p$	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	Buffer now full
	Ready		Ready	p5	Running	1	$T_{c1}$ awoken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Running		Sleep	1	$T_{c2}$ sneaks in ...
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	... and grabs data
	Ready	c5	Running		Ready	0	$T_p$ awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	Oh oh! No data

Figure 30.7: Thread Trace: Broken Solution (Version 1)

Let's examine the signaling logic between producers and consumers. When a producer wants to fill the buffer, it waits for it to be empty (p1–p3). The consumer has the exact same logic, but waits for a different condition: fullness (c1–c3).

With just a single producer and a single consumer, the code in Figure 30.6 works. However, if we have more than one of these threads (e.g., two consumers), the solution has two critical problems. What are they?

... (pause here to think) ...

Let's understand the first problem, which has to do with the `if` statement before the wait. Assume there are two consumers ( $T_{c1}$  and  $T_{c2}$ ) and one producer ( $T_p$ ). First, a consumer ( $T_{c1}$ ) runs; it acquires the lock (c1), checks if any buffers are ready for consumption (c2), and finding that none are, waits (c3) (which releases the lock).

Then the producer ( $T_p$ ) runs. It acquires the lock (p1), checks if all buffers are full (p2), and finding that not to be the case, goes ahead and fills the buffer (p4). The producer then signals that a buffer has been filled (p5). Critically, this moves the first consumer ( $T_{c1}$ ) from sleeping on a condition variable to the ready queue;  $T_{c1}$  is now able to run (but not yet running). The producer then continues until realizing the buffer is full, at which point it sleeps (p6, p1–p3).

Here is where the problem occurs: another consumer ( $T_{c2}$ ) sneaks in and consumes the one existing value in the buffer (c1, c2, c4, c5, c6, skipping the wait at c3 because the buffer is full). Now assume  $T_{c1}$  runs; just before returning from the wait, it re-acquires the lock and then returns. It then calls `get()` (c4), but there are no buffers to consume! An assertion triggers, and the code has not functioned as desired. Clearly, we should have somehow prevented  $T_{c1}$  from trying to consume because  $T_{c2}$  snuck in and consumed the one value in the buffer that had been produced. Figure 30.7 shows the action each thread takes, as well as its scheduler state (Ready, Running, or Sleeping) over time.

```

1 cond_t cond;
2 mutex_t mutex;
3
4 void *producer(void *arg) {
5     int i;
6     for (i = 0; i < loops; i++) {
7         Pthread_mutex_lock(&mutex);           // p1
8         while (count == 1)                  // p2
9             Pthread_cond_wait(&cond, &mutex); // p3
10        put(i);                         // p4
11        Pthread_cond_signal(&cond);       // p5
12        Pthread_mutex_unlock(&mutex);     // p6
13    }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);           // c1
20         while (count == 0)                  // c2
21             Pthread_cond_wait(&cond, &mutex); // c3
22         int tmp = get();                  // c4
23         Pthread_cond_signal(&cond);       // c5
24         Pthread_mutex_unlock(&mutex);     // c6
25         printf("%d\n", tmp);
26     }
27 }
```

Figure 30.8: Producer/Consumer: Single CV And While

The problem arises for a simple reason: after the producer woke  $T_{c1}$ , but *before*  $T_{c1}$  ever ran, the state of the bounded buffer changed (thanks to  $T_{c2}$ ). Signaling a thread only wakes them up; it is thus a *hint* that the state of the world has changed (in this case, that a value has been placed in the buffer), but there is no guarantee that when the woken thread runs, the state will *still* be as desired. This interpretation of what a signal means is often referred to as **Mesa semantics**, after the first research that built a condition variable in such a manner [LR80]; the contrast, referred to as **Hoare semantics**, is harder to build but provides a stronger guarantee that the woken thread will run immediately upon being woken [H74]. Virtually every system ever built employs Mesa semantics.

### Better, But Still Broken: While, Not If

Fortunately, this fix is easy (Figure 30.8): change the `if` to a `while`. Think about why this works; now consumer  $T_{c1}$  wakes up and (with the lock held) immediately re-checks the state of the shared variable (c2). If the buffer is empty at that point, the consumer simply goes back to sleep (c3). The corollary `if` is also changed to a `while` in the producer (p2).

Thanks to Mesa semantics, a simple rule to remember with condition variables is to **always use while loops**. Sometimes you don't have to re-check the condition, but it is always safe to do so; just do it and be happy.

$T_{c1}$	State	$T_{c2}$	State	$T_p$	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	$T_{c1}$ awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Running		Sleep		Sleep	1	Recheck condition
c4	Running		Sleep		Sleep	0	$T_{c1}$ grabs data
c5	Running		Ready		Sleep	0	Oops! Woke $T_{c2}$
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Running		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep...

Figure 30.9: Thread Trace: Broken Solution (Version 2)

However, this code still has a bug, the second of two problems mentioned above. Can you see it? It has something to do with the fact that there is only one condition variable. Try to figure out what the problem is, before reading ahead. DO IT!

... (another pause for you to think, or close your eyes for a bit) ...

Let's confirm you figured it out correctly, or perhaps let's confirm that you are now awake and reading this part of the book. The problem occurs when two consumers run first ( $T_{c1}$  and  $T_{c2}$ ), and both go to sleep (c3). Then, a producer runs, puts a value in the buffer, wakes one of the consumers (say  $T_{c1}$ ), and goes back to sleep. Now we have one consumer ready to run ( $T_{c1}$ ), and two threads sleeping on a condition ( $T_{c2}$  and  $T_p$ ). And we are about to cause a problem to occur: things are getting exciting!

The consumer  $T_{c1}$  then wakes by returning from `wait()` (c3), re-checks the condition (c2), and finding the buffer full, consumes the value (c4). This consumer then, critically, signals on the condition (c5), waking one thread that is sleeping. However, which thread should it wake?

Because the consumer has emptied the buffer, it clearly should wake the producer. However, if it wakes the consumer  $T_{c2}$  (which is definitely possible, depending on how the wait queue is managed), we have a problem. Specifically, the consumer  $T_{c2}$  will wake up and find the buffer empty (c2), and go back to sleep (c3). The producer  $T_p$ , which has a value to put into the buffer, is left sleeping. The other consumer thread,  $T_{c1}$ , also goes back to sleep. All three threads are left sleeping, a clear bug; see Figure 30.9 for the brutal step-by-step of this terrible calamity.

Signaling is clearly needed, but must be more directed. A consumer should not wake other consumers, only producers, and vice-versa.

```

1 cond_t empty, fill;
2 mutex_t mutex;
3
4 void *producer(void *arg) {
5     int i;
6     for (i = 0; i < loops; i++) {
7         Pthread_mutex_lock(&mutex);
8         while (count == 1)
9             Pthread_cond_wait(&empty, &mutex);
10        put(i);
11        Pthread_cond_signal(&fill);
12        Pthread_mutex_unlock(&mutex);
13    }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);
20         while (count == 0)
21             Pthread_cond_wait(&fill, &mutex);
22         int tmp = get();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

Figure 30.10: Producer/Consumer: Two CVs And While

## The Single Buffer Producer/Consumer Solution

The solution here is once again a small one: use *two* condition variables, instead of one, in order to properly signal which type of thread should wake up when the state of the system changes. Figure 30.10 shows the resulting code.

In the code above, producer threads wait on the condition **empty**, and signals **fill**. Conversely, consumer threads wait on **fill** and signal **empty**. By doing so, the second problem above is avoided by design: a consumer can never accidentally wake a consumer, and a producer can never accidentally wake a producer.

## The Final Producer/Consumer Solution

We now have a working producer/consumer solution, albeit not a fully general one. The last change we make is to enable more concurrency and efficiency; specifically, we add more buffer slots, so that multiple values can be produced before sleeping, and similarly multiple values can be consumed before sleeping. With just a single producer and consumer, this approach is more efficient as it reduces context switches; with multiple producers or consumers (or both), it even allows concurrent producing or consuming to take place, thus increasing concurrency. Fortunately, it is a small change from our current solution.

```

1  int buffer[MAX];
2  int fill_ptr = 0;
3  int use_ptr = 0;
4  int count = 0;
5
6  void put(int value) {
7      buffer[fill_ptr] = value;
8      fill_ptr = (fill_ptr + 1) % MAX;
9      count++;
10 }
11
12 int get() {
13     int tmp = buffer[use_ptr];
14     use_ptr = (use_ptr + 1) % MAX;
15     count--;
16     return tmp;
17 }
```

Figure 30.11: The Final Put And Get Routines

```

1 cond_t empty, fill;
2 mutex_t mutex;
3
4 void *producer(void *arg) {
5     int i;
6     for (i = 0; i < loops; i++) {
7         Pthread_mutex_lock(&mutex); // p1
8         while (count == MAX) // p2
9             Pthread_cond_wait(&empty, &mutex); // p3
10        put(i); // p4
11        Pthread_cond_signal(&fill); // p5
12        Pthread_mutex_unlock(&mutex); // p6
13    }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex); // c1
20         while (count == 0) // c2
21             Pthread_cond_wait(&fill, &mutex); // c3
22         int tmp = get(); // c4
23         Pthread_cond_signal(&empty); // c5
24         Pthread_mutex_unlock(&mutex); // c6
25         printf("%d\n", tmp);
26    }
27 }
```

Figure 30.12: The Final Working Solution

The first change for this final solution is within the buffer structure itself and the corresponding `put()` and `get()` (Figure 30.11). We also slightly change the conditions that producers and consumers check in order to determine whether to sleep or not. Figure 30.12 shows the final waiting and signaling logic. A producer only sleeps if all buffers are currently filled (p2); similarly, a consumer only sleeps if all buffers are currently empty (c2). And thus we solve the producer/consumer problem.

**TIP: USE WHILE (NOT IF) FOR CONDITIONS**

When checking for a condition in a multi-threaded program, using a `while` loop is always correct; using an `if` statement only might be, depending on the semantics of signaling. Thus, always use `while` and your code will behave as expected.

Using `while` loops around conditional checks also handles the case where **spurious wakeups** occur. In some thread packages, due to details of the implementation, it is possible that two threads get woken up though just a single signal has taken place [L11]. Spurious wakeups are further reason to re-check the condition a thread is waiting on.

### 30.3 Covering Conditions

We'll now look at one more example of how condition variables can be used. This code study is drawn from Lampson and Redell's paper on Pilot [LR80], the same group who first implemented the **Mesa semantics** described above (the language they used was Mesa, hence the name).

The problem they ran into is best shown via simple example, in this case in a simple multi-threaded memory allocation library. Figure 30.13 shows a code snippet which demonstrates the issue.

As you might see in the code, when a thread calls into the memory allocation code, it might have to wait in order for more memory to become free. Conversely, when a thread frees memory, it signals that more memory is free. However, our code above has a problem: which waiting thread (there can be more than one) should be woken up?

Consider the following scenario. Assume there are zero bytes free; thread  $T_a$  calls `allocate(100)`, followed by thread  $T_b$  which asks for less memory by calling `allocate(10)`. Both  $T_a$  and  $T_b$  thus wait on the condition and go to sleep; there aren't enough free bytes to satisfy either of these requests.

At that point, assume a third thread,  $T_c$ , calls `free(50)`. Unfortunately, when it calls `signal` to wake a waiting thread, it might not wake the correct waiting thread,  $T_b$ , which is waiting for only 10 bytes to be freed;  $T_a$  should remain waiting, as not enough memory is yet free. Thus, the code in the figure does not work, as the thread waking other threads does not know which thread (or threads) to wake up.

The solution suggested by Lampson and Redell is straightforward: replace the `pthread_cond_signal()` call in the code above with a call to `pthread_cond_broadcast()`, which wakes up *all* waiting threads. By doing so, we guarantee that any threads that should be woken are. The downside, of course, can be a negative performance impact, as we might needlessly wake up many other waiting threads that shouldn't (yet) be awake. Those threads will simply wake up, re-check the condition, and then go immediately back to sleep.

```

1 // how many bytes of the heap are free?
2 int bytesLeft = MAX_HEAP_SIZE;
3
4 // need lock and condition too
5 cond_t c;
6 mutex_t m;
7
8 void *
9 allocate(int size) {
10     Pthread_mutex_lock(&m);
11     while (bytesLeft < size)
12         Pthread_cond_wait(&c, &m);
13     void *ptr = ...; // get mem from heap
14     bytesLeft -= size;
15     Pthread_mutex_unlock(&m);
16     return ptr;
17 }
18
19 void free(void *ptr, int size) {
20     Pthread_mutex_lock(&m);
21     bytesLeft += size;
22     Pthread_cond_signal(&c); // whom to signal??
23     Pthread_mutex_unlock(&m);
24 }
```

Figure 30.13: Covering Conditions: An Example

Lampson and Redell call such a condition a **covering condition**, as it covers all the cases where a thread needs to wake up (conservatively); the cost, as we've discussed, is that too many threads might be woken. The astute reader might also have noticed we could have used this approach earlier (see the producer/consumer problem with only a single condition variable). However, in that case, a better solution was available to us, and thus we used it. In general, if you find that your program only works when you change your signals to broadcasts (but you don't think it should need to), you probably have a bug; fix it! But in cases like the memory allocator above, broadcast may be the most straightforward solution available.

## 30.4 Summary

We have seen the introduction of another important synchronization primitive beyond locks: condition variables. By allowing threads to sleep when some program state is not as desired, CVs enable us to neatly solve a number of important synchronization problems, including the famous (and still important) producer/consumer problem, as well as covering conditions. A more dramatic concluding sentence would go here, such as "He loved Big Brother" [O49].

## References

[D68] "Cooperating sequential processes"

Edsger W. Dijkstra, 1968

Available: <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF>

*Another classic from Dijkstra; reading his early works on concurrency will teach you much of what you need to know.*

[D72] "Information Streams Sharing a Finite Buffer"

E.W. Dijkstra

Information Processing Letters 1: 179180, 1972

Available: <http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD329.PDF>

*The famous paper that introduced the producer/consumer problem.*

[D01] "My recollections of operating system design"

E.W. Dijkstra

April, 2001

Available: <http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1303.PDF>

*A fascinating read for those of you interested in how the pioneers of our field came up with some very basic and fundamental concepts, including ideas like "interrupts" and even "a stack"!*

[H74] "Monitors: An Operating System Structuring Concept"

C.A.R. Hoare

Communications of the ACM, 17:10, pages 549–557, October 1974

*Hoare did a fair amount of theoretical work in concurrency. However, he is still probably most known for his work on Quicksort, the coolest sorting algorithm in the world, at least according to these authors.*

[L11] "Pthread\_cond\_signal Man Page"

Available: [http://linux.die.net/man/3/pthread\\_cond\\_signal](http://linux.die.net/man/3/pthread_cond_signal)

March, 2011

*The Linux man page shows a nice simple example of why a thread might get a spurious wakeup, due to race conditions within the signal/wakeup code.*

[LR80] "Experience with Processes and Monitors in Mesa"

B.W. Lampson, D.R. Redell

Communications of the ACM. 23:2, pages 105-117, February 1980

*A terrific paper about how to actually implement signaling and condition variables in a real system, leading to the term "Mesa" semantics for what it means to be woken up; the older semantics, developed by Tony Hoare [H74], then became known as "Hoare" semantics, which is hard to say out loud in class with a straight face.*

[O49] "1984"

George Orwell, 1949, Secker and Warburg

*A little heavy-handed, but of course a must read. That said, we kind of gave away the ending by quoting the last sentence. Sorry! And if the government is reading this, let us just say that we think that the government is "double plus good". Hear that, our pals at the NSA?*

## Homework

This homework lets you explore some real code that uses locks and condition variables to implement various forms of the producer/consumer queue discussed in the chapter. You'll look at the real code, run it in various configurations, and use it to learn about what works and what doesn't, as well as other intricacies.

The different versions of the code correspond to different ways to "solve" the producer/consumer problem. Most are incorrect; one is correct. Read the chapter to learn more about what the producer/consumer problem is, and what the code generally does.

The first step is to download the code and type `make` to build all the variants. You should see four:

- `main-one-cv-while.c`: The producer/consumer problem solved with a single condition variable.
- `main-two-cvs-if.c`: Same but with two condition variables and using an `if` to check whether to sleep.
- `main-two-cvs-while.c`: Same but with two condition variables and `while` to check whether to sleep. **This is the correct version.**
- `main-two-cvs-while-extra-unlock.c`: Same but releasing the lock and then reacquiring it around the fill and get routines.

It's also useful to look at `pc-header.h` which contains common code for all of these different main programs, and the `Makefile` so as to build the code properly.

See the `README` for details on these programs.

## Questions

1. Our first question focuses on `main-two-cvs-while.c` (the working solution). First, study the code. Do you think you have an understanding of what should happen when you run the program?
2. Now run with one producer and one consumer, and have the producer produce a few values. Start with a buffer of size 1, and then increase it. How does the behavior of the code change when the buffer is larger? (or does it?) What would you predict `num_full` to be with different buffer sizes (e.g., `-m 10`) and different numbers of produced items (e.g., `-l 100`), when you change the consumer sleep string from default (no sleep) to `-C 0,0,0,0,0,0,1`?
3. If possible, run the code on different systems (e.g., Mac OS X and Linux). Do you see different behavior across these systems?

4. Let's look at some timings of different runs. How long do you think the following execution, with one producer, three consumers, a single-entry shared buffer, and each consumer pausing at point c3 for a second, will take?

```
prompt> ./main-one-cv-while -p 1 -c 3 -m 1 -c  
0,0,0,1,0,0,0:0,0,1,0,0,0:0,0,0,1,0,0,0 -l 10 -v -t
```

5. Now change the size of the shared buffer to 3 (-m 3). Will this make any difference in the total time?
6. Now change the location of the sleep to c6 (this models a consumer taking something off the queue and then doing something with it for a while), again using a single-entry buffer. What time do you predict in this case?

```
prompt> ./main-one-cv-while -p 1 -c 3 -m 1 -c  
0,0,0,0,0,0,1:0,0,0,0,0,1:0,0,0,0,0,1 -l 10 -v -t
```

7. Finally, change the buffer size to 3 again (-m 3). What time do you predict now?
8. Now let's look at `main-one-cv-while.c`. Can you configure a sleep string, assuming a single producer, one consumer, and a buffer of size 1, to cause a problem with this code?
9. Now change the number of consumers to two. Can you construct sleep strings for the producer and the consumers so as to cause a problem in the code?
10. Now examine `main-two-cvs-if.c`. Can you cause a problem to happen in this code? Again consider the case where there is only one consumer, and then the case where there is more than one.
11. Finally, examine `main-two-cvs-while-extra-unlock.c`. What problem arises when you release the lock before doing a put or a get? Can you reliably cause such a problem to happen, given the sleep strings? What bad thing can happen?

## Semaphores

As we know now, one needs both locks and condition variables to solve a broad range of relevant and interesting concurrency problems. One of the first people to realize this years ago was **Edsger Dijkstra** (though it is hard to know the exact history [GR92]), known among other things for his famous “shortest paths” algorithm in graph theory [D59], an early polemic on structured programming entitled “Goto Statements Considered Harmful” [D68a] (what a great title!), and, in the case we will study here, the introduction of a synchronization primitive called the **semaphore** [D68b,D72]. Indeed, Dijkstra and colleagues invented the semaphore as a single primitive for all things related to synchronization; as you will see, one can use semaphores as both locks and condition variables.

### THE CRUX: HOW TO USE SEMAPHORES

How can we use semaphores instead of locks and condition variables? What is the definition of a semaphore? What is a binary semaphore? Is it straightforward to build a semaphore out of locks and condition variables? To build locks and condition variables out of semaphores?

### 31.1 Semaphores: A Definition

A semaphore is an object with an integer value that we can manipulate with two routines; in the POSIX standard, these routines are `sem_wait()` and `sem_post()`<sup>1</sup>. Because the initial value of the semaphore determines its behavior, before calling any other routine to interact with the semaphore, we must first initialize it to some value, as the code in Figure 31.1 does.

---

<sup>1</sup>Historically, `sem_wait()` was called P() by Dijkstra and `sem_post()` called V(). P() comes from “prolaag”, a contraction of “probeer” (Dutch for “try”) and “verlaag” (“decrease”); V() comes from the Dutch word “verhoog” which means “increase” (thanks to Mart Oskamp for this information). Sometimes, people call them down and up. Use the Dutch versions to impress your friends, or confuse them, or both.

```

1 #include <semaphore.h>
2 sem_t s;
3 sem_init(&s, 0, 1);

```

Figure 31.1: Initializing A Semaphore

In the figure, we declare a semaphore `s` and initialize it to the value 1 by passing 1 in as the third argument. The second argument to `sem_init()` will be set to 0 in all of the examples we'll see; this indicates that the semaphore is shared between threads in the same process. See the man page for details on other usages of semaphores (namely, how they can be used to synchronize access across *different* processes), which require a different value for that second argument.

After a semaphore is initialized, we can call one of two functions to interact with it, `sem_wait()` or `sem_post()`. The behavior of these two functions is seen in Figure 31.2.

For now, we are not concerned with the implementation of these routines, which clearly requires some care; with multiple threads calling into `sem_wait()` and `sem_post()`, there is the obvious need for managing these critical sections. We will now focus on how to *use* these primitives; later we may discuss how they are built.

We should discuss a few salient aspects of the interfaces here. First, we can see that `sem_wait()` will either return right away (because the value of the semaphore was one or higher when we called `sem_wait()`), or it will cause the caller to suspend execution waiting for a subsequent post. Of course, multiple calling threads may call into `sem_wait()`, and thus all be queued waiting to be woken.

Second, we can see that `sem_post()` does not wait for some particular condition to hold like `sem_wait()` does. Rather, it simply increments the value of the semaphore and then, if there is a thread waiting to be woken, wakes one of them up.

Third, the value of the semaphore, when negative, is equal to the number of waiting threads [D68b]. Though the value generally isn't seen by users of the semaphores, this invariant is worth knowing and perhaps can help you remember how a semaphore functions.

Don't worry (yet) about the seeming race conditions possible within the semaphore; assume that the actions they make are performed atomically. We will soon use locks and condition variables to do just this.

```

1 int sem_wait(sem_t *s) {
2     decrement the value of semaphore s by one
3     wait if value of semaphore s is negative
4 }
5
6 int sem_post(sem_t *s) {
7     increment the value of semaphore s by one
8     if there are one or more threads waiting, wake one
9 }

```

Figure 31.2: Semaphore: Definitions Of Wait And Post

```

1 sem_t m;
2 sem_init(&m, 0, X); // initialize semaphore to X; what should X be?
3
4 sem_wait(&m);
5 // critical section here
6 sem_post(&m);

```

Figure 31.3: A Binary Semaphore (That Is, A Lock)

## 31.2 Binary Semaphores (Locks)

We are now ready to use a semaphore. Our first use will be one with which we are already familiar: using a semaphore as a lock. See Figure 31.3 for a code snippet; therein, you'll see that we simply surround the critical section of interest with a `sem_wait()`/`sem_post()` pair. Critical to making this work, though, is the initial value of the semaphore `m` (initialized to `X` in the figure). What should `X` be?

*... (Try thinking about it before going on) ...*

Looking back at definition of the `sem_wait()` and `sem_post()` routines above, we can see that the initial value should be 1.

To make this clear, let's imagine a scenario with two threads. The first thread (Thread 0) calls `sem_wait()`; it will first decrement the value of the semaphore, changing it to 0. Then, it will wait only if the value is *not* greater than or equal to 0. Because the value is 0, `sem_wait()` will simply return and the calling thread will continue; Thread 0 is now free to enter the critical section. If no other thread tries to acquire the lock while Thread 0 is inside the critical section, when it calls `sem_post()`, it will simply restore the value of the semaphore to 1 (and not wake a waiting thread, because there are none). Figure 31.4 shows a trace of this scenario.

A more interesting case arises when Thread 0 "holds the lock" (i.e., it has called `sem_wait()` but not yet called `sem_post()`), and another thread (Thread 1) tries to enter the critical section by calling `sem_wait()`. In this case, Thread 1 will decrement the value of the semaphore to -1, and thus wait (putting itself to sleep and relinquishing the processor). When Thread 0 runs again, it will eventually call `sem_post()`, incrementing the value of the semaphore back to zero, and then wake the waiting thread (Thread 1), which will then be able to acquire the lock for itself. When Thread 1 finishes, it will again increment the value of the semaphore, restoring it to 1 again.

Value of Semaphore	Thread 0	Thread 1
1		
1	call <code>sem_wait()</code>	
0	<code>sem.wait()</code> returns	
0	(crit sect)	
0	call <code>sem_post()</code>	
1	<code>sem.post()</code> returns	

Figure 31.4: Thread Trace: Single Thread Using A Semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call sem_wait()	Running		Ready
0	sem_wait() returns	Running		Ready
0	(crit sect: begin)	Running		Ready
0	Interrupt; Switch→T1	Ready		Running
0		Ready	call sem_wait()	Running
-1		Ready	decrement sem	Running
-1		Ready	(sem<0) → sleep	Sleeping
-1		Running	Switch→T0	Sleeping
-1	(crit sect: end)	Running		Sleeping
-1	call sem_post()	Running		Sleeping
0	increment sem	Running		Sleeping
0	wake(T1)	Running		Ready
0	sem_post() returns	Running		Ready
0	Interrupt; Switch→T1	Ready	sem_wait() returns	Running
0		Ready	(crit sect)	Running
0		Ready	call sem_post()	Running
1		Ready	sem_post() returns	Running

Figure 31.5: Thread Trace: Two Threads Using A Semaphore

Figure 31.5 shows a trace of this example. In addition to thread actions, the figure shows the **scheduler state** of each thread: Running, Ready (i.e., runnable but not running), and Sleeping. Note in particular that Thread 1 goes into the sleeping state when it tries to acquire the already-held lock; only when Thread 0 runs again can Thread 1 be awoken and potentially run again.

If you want to work through your own example, try a scenario where multiple threads queue up waiting for a lock. What would the value of the semaphore be during such a trace?

Thus we are able to use semaphores as locks. Because locks only have two states (held and not held), we sometimes call a semaphore used as a lock a **binary semaphore**. Note that if you are using a semaphore only in this binary fashion, it could be implemented in a simpler manner than the generalized semaphores we present here.

### 31.3 Semaphores As Condition Variables

Semaphores are also useful when a thread wants to halt its progress waiting for a condition to become true. For example, a thread may wish to wait for a list to become non-empty, so it can delete an element from it. In this pattern of usage, we often find a thread *waiting* for something to happen, and a different thread making that something happen and then *signaling* that it has happened, thus waking the waiting thread. Because the waiting thread (or threads) is waiting for some **condition** in the program to change, we are using the semaphore as a **condition variable**.

```

1 sem_t s;
2
3 void *
4 child(void *arg) {
5     printf("child\n");
6     sem_post(&s); // signal here: child is done
7     return NULL;
8 }
9
10 int
11 main(int argc, char *argv[]) {
12     sem_init(&s, 0, X); // what should X be?
13     printf("parent: begin\n");
14     pthread_t c;
15     Pthread_create(c, NULL, child, NULL);
16     sem_wait(&s); // wait here for child
17     printf("parent: end\n");
18     return 0;
19 }
```

Figure 31.6: A Parent Waiting For Its Child

A simple example is as follows. Imagine a thread creates another thread and then wants to wait for it to complete its execution (Figure 31.6). When this program runs, we would like to see the following:

```
parent: begin
child
parent: end
```

The question, then, is how to use a semaphore to achieve this effect; as it turns out, the answer is relatively easy to understand. As you can see in the code, the parent simply calls `sem_wait()` and the child `sem_post()` to wait for the condition of the child finishing its execution to become true. However, this raises the question: what should the initial value of this semaphore be?

*(Again, think about it here, instead of reading ahead)*

The answer, of course, is that the value of the semaphore should be set to 0. There are two cases to consider. First, let us assume that the parent creates the child but the child has not run yet (i.e., it is sitting in a ready queue but not running). In this case (Figure 31.7, page 6), the parent will call `sem_wait()` before the child has called `sem_post()`; we'd like the parent to wait for the child to run. The only way this will happen is if the value of the semaphore is not greater than 0; hence, 0 is the initial value. The parent runs, decrements the semaphore (to -1), then waits (sleeping). When the child finally runs, it will call `sem_post()`, increment the value of the semaphore to 0, and wake the parent, which will then return from `sem_wait()` and finish the program.

The second case (Figure 31.8, page 6) occurs when the child runs to completion before the parent gets a chance to call `sem_wait()`. In this case, the child will first call `sem_post()`, thus incrementing the value of the semaphore from 0 to 1. When the parent then gets a chance to run, it will call `sem_wait()` and find the value of the semaphore to be 1; the parent will thus decrement the value (to 0) and return from `sem_wait()` without waiting, also achieving the desired effect.

Value	Parent	State	Child	State
0	create(Child)	Running	(Child exists; is runnable)	Ready
0	call sem_wait()	Running		Ready
-1	decrement sem	Running		Ready
-1	(sem<0) → sleep	Sleeping		Ready
-1	Switch→Child	Sleeping	child runs	Running
-1		Sleeping	call sem_post()	Running
0		Sleeping	increment sem	Running
0		Ready	wake(Parent)	Running
0		Ready	sem_post() returns	Running
0		Ready	Interrupt; Switch→Parent	Ready
0	sem_wait() returns	Running		Ready

Figure 31.7: Thread Trace: Parent Waiting For Child (Case 1)

Value	Parent	State	Child	State
0	create(Child)	Running	(Child exists; is runnable)	Ready
0	Interrupt; Switch→Child	Ready	child runs	Running
0		Ready	call sem_post()	Running
1		Ready	increment sem	Running
1		Ready	wake(nobody)	Running
1		Ready	sem_post() returns	Running
1	parent runs	Running	Interrupt; Switch→Parent	Ready
1	call sem_wait()	Running		Ready
0	decrement sem	Running		Ready
0	(sem≥0) → awake	Running		Ready
0	sem_wait() returns	Running		Ready

Figure 31.8: Thread Trace: Parent Waiting For Child (Case 2)

## 31.4 The Producer/Consumer (Bounded Buffer) Problem

The next problem we will confront in this chapter is known as the **producer/consumer** problem, or sometimes as the **bounded buffer** problem [D72]. This problem is described in detail in the previous chapter on condition variables; see there for details.

### First Attempt

Our first attempt at solving the problem introduces two semaphores, `empty` and `full`, which the threads will use to indicate when a buffer entry has been emptied or filled, respectively. The code for the put and get routines is in Figure 31.9, and our attempt at solving the producer and consumer problem is in Figure 31.10.

In this example, the producer first waits for a buffer to become empty in order to put data into it, and the consumer similarly waits for a buffer to become filled before using it. Let us first imagine that `MAX=1` (there is only one buffer in the array), and see if this works.

Imagine again there are two threads, a producer and a consumer. Let us examine a specific scenario on a single CPU. Assume the consumer gets to run first. Thus, the consumer will hit line `c1` in the figure above, calling `sem_wait(&full)`. Because `full` was initialized to the value 0,

```

1 int buffer[MAX];
2 int fill = 0;
3 int use = 0;
4
5 void put(int value) {
6     buffer[fill] = value;      // line f1
7     fill = (fill + 1) % MAX; // line f2
8 }
9
10 int get() {
11     int tmp = buffer[use];    // line g1
12     use = (use + 1) % MAX;   // line g2
13     return tmp;
14 }
```

Figure 31.9: The Put And Get Routines

```

1 sem_t empty;
2 sem_t full;
3
4 void *producer(void *arg) {
5     int i;
6     for (i = 0; i < loops; i++) {
7         sem_wait(&empty);           // line P1
8         put(i);                  // line P2
9         sem_post(&full);          // line P3
10    }
11 }
12
13 void *consumer(void *arg) {
14     int i, tmp = 0;
15     while (tmp != -1) {
16         sem_wait(&full);        // line C1
17         tmp = get();           // line C2
18         sem_post(&empty);       // line C3
19         printf("%d\n", tmp);
20     }
21 }
22
23 int main(int argc, char *argv[]) {
24     // ...
25     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
26     sem_init(&full, 0, 0);   // ... and 0 are full
27     // ...
28 }
```

Figure 31.10: Adding The Full And Empty Conditions

the call will decrement `full` (to -1), block the consumer, and wait for another thread to call `sem_post()` on `full`, as desired.

Assume the producer then runs. It will hit line P1, thus calling the `sem_wait(&empty)` routine. Unlike the consumer, the producer will continue through this line, because `empty` was initialized to the value MAX (in this case, 1). Thus, `empty` will be decremented to 0 and the producer will put a data value into the first entry of `buffer` (line P2). The producer will then continue on to P3 and call `sem_post (&full)`, changing the value of the full semaphore from -1 to 0 and waking the consumer (e.g., move it from blocked to ready).

In this case, one of two things could happen. If the producer continues to run, it will loop around and hit line P1 again. This time, however, it would block, as the empty semaphore's value is 0. If the producer instead was interrupted and the consumer began to run, it would call `sem_wait(&full)` (line c1) and find that the buffer was indeed full and thus consume it. In either case, we achieve the desired behavior.

You can try this same example with more threads (e.g., multiple producers, and multiple consumers). It should still work.

Let us now imagine that MAX is greater than 1 (say MAX = 10). For this example, let us assume that there are multiple producers and multiple consumers. We now have a problem: a race condition. Do you see where it occurs? (take some time and look for it) If you can't see it, here's a hint: look more closely at the `put()` and `get()` code.

OK, let's understand the issue. Imagine two producers (Pa and Pb) both calling into `put()` at roughly the same time. Assume producer Pa gets to run first, and just starts to fill the first buffer entry (`fill = 0` at line f1). Before Pa gets a chance to increment the fill counter to 1, it is interrupted. Producer Pb starts to run, and at line f1 it also puts its data into the 0th element of buffer, which means that the old data there is overwritten! This is a no-no; we don't want any data from the producer to be lost.

## A Solution: Adding Mutual Exclusion

As you can see, what we've forgotten here is *mutual exclusion*. The filling of a buffer and incrementing of the index into the buffer is a critical section, and thus must be guarded carefully. So let's use our friend the binary semaphore and add some locks. Figure 31.11 shows our attempt.

Now we've added some locks around the entire `put()`/`get()` parts of the code, as indicated by the NEW LINE comments. That seems like the right idea, but it also doesn't work. Why? Deadlock. Why does deadlock occur? Take a moment to consider it; try to find a case where deadlock arises. What sequence of steps must happen for the program to deadlock?

## Avoiding Deadlock

OK, now that you figured it out, here is the answer. Imagine two threads, one producer and one consumer. The consumer gets to run first. It acquires the mutex (line c0), and then calls `sem_wait()` on the full semaphore (line c1); because there is no data yet, this call causes the consumer to block and thus yield the CPU; importantly, though, the consumer still holds the lock.

A producer then runs. It has data to produce and if it were able to run, it would be able to wake the consumer thread and all would be good. Unfortunately, the first thing it does is call `sem_wait()` on the binary mutex semaphore (line p0). The lock is already held. Hence, the producer is now stuck waiting too.

```

1 sem_t empty;
2 sem_t full;
3 sem_t mutex;
4
5 void *producer(void *arg) {
6     int i;
7     for (i = 0; i < loops; i++) {
8         sem_wait(&mutex);           // line p0 (NEW LINE)
9         sem_wait(&empty);          // line p1
10        put(i);                  // line p2
11        sem_post(&full);          // line p3
12        sem_post(&mutex);          // line p4 (NEW LINE)
13    }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&mutex);           // line c0 (NEW LINE)
20         sem_wait(&full);           // line c1
21         int tmp = get();           // line c2
22         sem_post(&empty);          // line c3
23         sem_post(&mutex);          // line c4 (NEW LINE)
24         printf("%d\n", tmp);
25     }
26 }
27
28 int main(int argc, char *argv[]) {
29     // ...
30     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31     sem_init(&full, 0, 0);    // ... and 0 are full
32     sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock (NEW LINE)
33     // ...
34 }
```

Figure 31.11: Adding Mutual Exclusion (Incorrectly)

There is a simple cycle here. The consumer *holds* the mutex and is *waiting* for the someone to signal full. The producer could *signal* full but is *waiting* for the mutex. Thus, the producer and consumer are each stuck waiting for each other: a classic deadlock.

## Finally, A Working Solution

To solve this problem, we simply must reduce the scope of the lock. Figure 31.12 shows the final working solution. As you can see, we simply move the mutex acquire and release to be just around the critical section; the full and empty wait and signal code is left outside. The result is a simple and working bounded buffer, a commonly-used pattern in multi-threaded programs. Understand it now; use it later. You will thank us for years to come. Or at least, you will thank us when the same question is asked on the final exam.

```

1  sem_t empty;
2  sem_t full;
3  sem_t mutex;
4
5  void *producer(void *arg) {
6      int i;
7      for (i = 0; i < loops; i++) {
8          sem_wait(&empty);           // line p1
9          sem_wait(&mutex);         // line p1.5 (MOVED MUTEX HERE...)
10         put(i);                // line p2
11         sem_post(&mutex);        // line p2.5 (... AND HERE)
12         sem_post(&full);         // line p3
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&full);          // line c1
20         sem_wait(&mutex);         // line c1.5 (MOVED MUTEX HERE...)
21         int tmp = get();          // line c2
22         sem_post(&mutex);        // line c2.5 (... AND HERE)
23         sem_post(&empty);         // line c3
24         printf("%d\n", tmp);
25     }
26 }
27
28 int main(int argc, char *argv[]) {
29     // ...
30     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31     sem_init(&full, 0, 0);    // ... and 0 are full
32     sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
33     // ...
34 }
```

Figure 31.12: Adding Mutual Exclusion (Correctly)

## 31.5 Reader-Writer Locks

Another classic problem stems from the desire for a more flexible locking primitive that admits that different data structure accesses might require different kinds of locking. For example, imagine a number of concurrent list operations, including inserts and simple lookups. While inserts change the state of the list (and thus a traditional critical section makes sense), lookups simply *read* the data structure; as long as we can guarantee that no insert is on-going, we can allow many lookups to proceed concurrently. The special type of lock we will now develop to support this type of operation is known as a **reader-writer lock** [CHP71]. The code for such a lock is available in Figure 31.13.

The code is pretty simple. If some thread wants to update the data structure in question, it should call the new pair of synchronization operations: `rwlock_acquire_writelock()`, to acquire a write lock, and `rwlock_release_writelock()`, to release it. Internally, these simply use the `writelock` semaphore to ensure that only a single writer can ac-

```

1  typedef struct _rwlock_t {
2      sem_t lock;          // binary semaphore (basic lock)
3      sem_t writelock;    // used to allow ONE writer or MANY readers
4      int readers;        // count of readers reading in critical section
5  } rwlock_t;
6
7  void rwlock_init(rwlock_t *rw) {
8      rw->readers = 0;
9      sem_init(&rw->lock, 0, 1);
10     sem_init(&rw->writelock, 0, 1);
11 }
12
13 void rwlock_acquire_readlock(rwlock_t *rw) {
14     sem_wait(&rw->lock);
15     rw->readers++;
16     if (rw->readers == 1)
17         sem_wait(&rw->writelock); // first reader acquires writelock
18     sem_post(&rw->lock);
19 }
20
21 void rwlock_release_readlock(rwlock_t *rw) {
22     sem_wait(&rw->lock);
23     rw->readers--;
24     if (rw->readers == 0)
25         sem_post(&rw->writelock); // last reader releases writelock
26     sem_post(&rw->lock);
27 }
28
29 void rwlock_acquire_writelock(rwlock_t *rw) {
30     sem_wait(&rw->writelock);
31 }
32
33 void rwlock_release_writelock(rwlock_t *rw) {
34     sem_post(&rw->writelock);
35 }
```

Figure 31.13: A Simple Reader-Writer Lock

quire the lock and thus enter the critical section to update the data structure in question.

More interesting is the pair of routines to acquire and release read locks. When acquiring a read lock, the reader first acquires `lock` and then increments the `readers` variable to track how many readers are currently inside the data structure. The important step then taken within `rwlock_acquire_readlock()` occurs when the first reader acquires the lock; in that case, the reader also acquires the write lock by calling `sem_wait()` on the `writelock` semaphore, and then finally releasing the `lock` by calling `sem_post()`.

Thus, once a reader has acquired a read lock, more readers will be allowed to acquire the read lock too; however, any thread that wishes to acquire the write lock will have to wait until *all* readers are finished; the last one to exit the critical section calls `sem_post()` on “`writelock`” and thus enables a waiting writer to acquire the lock.

This approach works (as desired), but does have some negatives, espe-

**TIP: SIMPLE AND DUMB CAN BE BETTER (HILL'S LAW)**

You should never underestimate the notion that the simple and dumb approach can be the best one. With locking, sometimes a simple spin lock works best, because it is easy to implement and fast. Although something like reader/writer locks sounds cool, they are complex, and complex can mean slow. Thus, always try the simple and dumb approach first.

This idea, of appealing to simplicity, is found in many places. One early source is Mark Hill's dissertation [H87], which studied how to design caches for CPUs. Hill found that simple direct-mapped caches worked better than fancy set-associative designs (one reason is that in caching, simpler designs enable faster lookups). As Hill succinctly summarized his work: "Big and dumb is better." And thus we call this similar advice **Hill's Law**.

cially when it comes to fairness. In particular, it would be relatively easy for readers to starve writers. More sophisticated solutions to this problem exist; perhaps you can think of a better implementation? Hint: think about what you would need to do to prevent more readers from entering the lock once a writer is waiting.

Finally, it should be noted that reader-writer locks should be used with some caution. They often add more overhead (especially with more sophisticated implementations), and thus do not end up speeding up performance as compared to just using simple and fast locking primitives [CB08]. Either way, they showcase once again how we can use semaphores in an interesting and useful way.

### 31.6 The Dining Philosophers

One of the most famous concurrency problems posed, and solved, by Dijkstra, is known as the **dining philosopher's problem** [D71]. The problem is famous because it is fun and somewhat intellectually interesting; however, its practical utility is low. However, its fame forces its inclusion here; indeed, you might be asked about it on some interview, and you'd really hate your OS professor if you miss that question and don't get the job. Conversely, if you get the job, please feel free to send your OS professor a nice note, or some stock options.

The basic setup for the problem is this (as shown in Figure 31.14): assume there are five "philosophers" sitting around a table. Between each pair of philosophers is a single fork (and thus, five total). The philosophers each have times where they think, and don't need any forks, and times where they eat. In order to eat, a philosopher needs two forks, both the one on their left and the one on their right. The contention for these forks, and the synchronization problems that ensue, are what makes this a problem we study in concurrent programming.

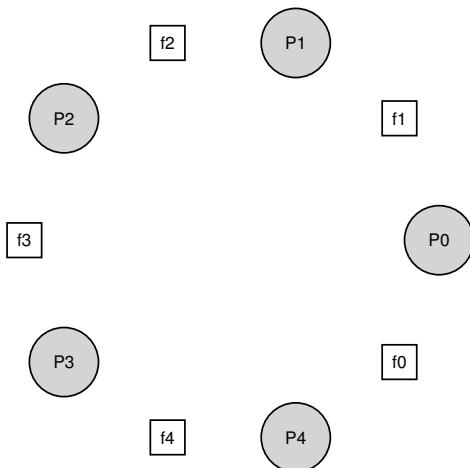


Figure 31.14: The Dining Philosophers

Here is the basic loop of each philosopher:

```
while (1) {
    think();
    getforks();
    eat();
    putforks();
}
```

The key challenge, then, is to write the routines `getforks()` and `putforks()` such that there is no deadlock, no philosopher starves and never gets to eat, and concurrency is high (i.e., as many philosophers can eat at the same time as possible).

Following Downey's solutions [D08], we'll use a few helper functions to get us towards a solution. They are:

```
int left(int p) { return p; }
int right(int p) { return (p + 1) % 5; }
```

When philosopher  $p$  wishes to refer to the fork on their left, they simply call `left(p)`. Similarly, the fork on the right of a philosopher  $p$  is referred to by calling `right(p)`; the modulo operator therein handles the one case where the last philosopher ( $p=4$ ) tries to grab the fork on their right, which is fork 0.

We'll also need some semaphores to solve this problem. Let us assume we have five, one for each fork: `sem_t forks[5]`.

```

1 void getforks() {
2     sem_wait(forks[left(p)]);
3     sem_wait(forks[right(p)]);
4 }
5
6 void putforks() {
7     sem_post(forks[left(p)]);
8     sem_post(forks[right(p)]);
9 }
```

Figure 31.15: The `getforks()` And `putforks()` Routines

### Broken Solution

We attempt our first solution to the problem. Assume we initialize each semaphore (in the `forks` array) to a value of 1. Assume also that each philosopher knows its own number (`p`). We can thus write the `getforks()` and `putforks()` routine as shown in Figure 31.15.

The intuition behind this (broken) solution is as follows. To acquire the forks, we simply grab a “lock” on each one: first the one on the left, and then the one on the right. When we are done eating, we release them. Simple, no? Unfortunately, in this case, simple means broken. Can you see the problem that arises? Think about it.

The problem is **deadlock**. If each philosopher happens to grab the fork on their left before any philosopher can grab the fork on their right, each will be stuck holding one fork and waiting for another, forever. Specifically, philosopher 0 grabs fork 0, philosopher 1 grabs fork 1, philosopher 2 grabs fork 2, philosopher 3 grabs fork 3, and philosopher 4 grabs fork 4; all the forks are acquired, and all the philosophers are stuck waiting for a fork that another philosopher possesses. We’ll study deadlock in more detail soon; for now, it is safe to say that this is not a working solution.

### A Solution: Breaking The Dependency

The simplest way to attack this problem is to change how forks are acquired by at least one of the philosophers; indeed, this is how Dijkstra himself solved the problem. Specifically, let’s assume that philosopher 4 (the highest numbered one) acquires the forks in a *different* order. The code to do so is as follows:

```

1 void getforks() {
2     if (p == 4) {
3         sem_wait(forks[right(p)]);
4         sem_wait(forks[left(p)]);
5     } else {
6         sem_wait(forks[left(p)]);
7         sem_wait(forks[right(p)]);
8     }
9 }
```

Because the last philosopher tries to grab right before left, there is no situation where each philosopher grabs one fork and is stuck waiting for another; the cycle of waiting is broken. Think through the ramifications of this solution, and convince yourself that it works.

```

1  typedef struct __Zem_t {
2      int value;
3      pthread_cond_t cond;
4      pthread_mutex_t lock;
5  } Zem_t;
6
7  // only one thread can call this
8  void Zem_init(Zem_t *s, int value) {
9      s->value = value;
10     Cond_init(&s->cond);
11     Mutex_init(&s->lock);
12 }
13
14 void Zem_wait(Zem_t *s) {
15     Mutex_lock(&s->lock);
16     while (s->value <= 0)
17         Cond_wait(&s->cond, &s->lock);
18     s->value--;
19     Mutex_unlock(&s->lock);
20 }
21
22 void Zem_post(Zem_t *s) {
23     Mutex_lock(&s->lock);
24     s->value++;
25     Cond_signal(&s->cond);
26     Mutex_unlock(&s->lock);
27 }
```

Figure 31.16: Implementing Zemaphores With Locks And CVs

There are other “famous” problems like this one, e.g., the **cigarette smoker’s problem** or the **sleeping barber problem**. Most of them are just excuses to think about concurrency; some of them have fascinating names. Look them up if you are interested in learning more, or just getting more practice thinking in a concurrent manner [D08].

### 31.7 How To Implement Semaphores

Finally, let’s use our low-level synchronization primitives, locks and condition variables, to build our own version of semaphores called ... (*drum roll here*) ... **Zemaphores**. This task is fairly straightforward, as you can see in Figure 31.16.

As you can see from the figure, we use just one lock and one condition variable, plus a state variable to track the value of the semaphore. Study the code for yourself until you really understand it. Do it!

One subtle difference between our Zemaphore and pure semaphores as defined by Dijkstra is that we don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads; indeed, the value will never be lower than zero. This behavior is easier to implement and matches the current Linux implementation.

Curiously, building locks and condition variables out of semaphores

**TIP: BE CAREFUL WITH GENERALIZATION**

The abstract technique of generalization can thus be quite useful in systems design, where one good idea can be made slightly broader and thus solve a larger class of problems. However, be careful when generalizing; as Lampson warns us “Don’t generalize; generalizations are generally wrong” [L83].

One could view semaphores as a generalization of locks and condition variables; however, is such a generalization needed? And, given the difficulty of realizing a condition variable on top of a semaphore, perhaps this generalization is not as general as you might think.

is a much trickier proposition. Some highly experienced concurrent programmers tried to do this in the Windows environment, and many different bugs ensued [B04]. Try it yourself, and see if you can figure out why building condition variables out of semaphores is more challenging than it might appear.

## 31.8 Summary

Semaphores are a powerful and flexible primitive for writing concurrent programs. Some programmers use them exclusively, shunning locks and condition variables, due to their simplicity and utility.

In this chapter, we have presented just a few classic problems and solutions. If you are interested in finding out more, there are many other materials you can reference. One great (and free reference) is Allen Downey’s book on concurrency and programming with semaphores [D08]. This book has lots of puzzles you can work on to improve your understanding of both semaphores in specific and concurrency in general. Becoming a real concurrency expert takes years of effort; going beyond what you learn in this class is undoubtedly the key to mastering such a topic.

## References

[B04] "Implementing Condition Variables with Semaphores"

Andrew Birrell

December 2004

*An interesting read on how difficult implementing CVs on top of semaphores really is, and the mistakes the author and co-workers made along the way. Particularly relevant because the group had done a ton of concurrent programming; Birrell, for example, is known for (among other things) writing various thread-programming guides.*

[CB08] "Real-world Concurrency"

Bryan Cantrill and Jeff Bonwick

ACM Queue. Volume 6, No. 5. September 2008

*A nice article by some kernel hackers from a company formerly known as Sun on the real problems faced in concurrent code.*

[CHP71] "Concurrent Control with Readers and Writers"

P.J. Courtois, F. Heymans, D.L. Parnas

Communications of the ACM, 14:10, October 1971

*The introduction of the reader-writer problem, and a simple solution. Later work introduced more complex solutions, skipped here because, well, they are pretty complex.*

[D59] "A Note on Two Problems in Connexion with Graphs"

E. W. Dijkstra

Numerische Mathematik 1, 269271, 1959

Available: <http://www-m3.ma.tum.de/twiki/pub/MN0506/WebHome/dijkstra.pdf>

*Can you believe people worked on algorithms in 1959? We can't. Even before computers were any fun to use, these people had a sense that they would transform the world...*

[D68a] "Go-to Statement Considered Harmful"

E.W. Dijkstra

Communications of the ACM, volume 11(3): pages 147148, March 1968

Available: <http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF>

*Sometimes thought as the beginning of the field of software engineering.*

[D68b] "The Structure of the THE Multiprogramming System"

E.W. Dijkstra

Communications of the ACM, volume 11(5), pages 341346, 1968

*One of the earliest papers to point out that systems work in computer science is an engaging intellectual endeavor. Also argues strongly for modularity in the form of layered systems.*

[D72] "Information Streams Sharing a Finite Buffer"

E.W. Dijkstra

Information Processing Letters 1: 179180, 1972

Available: <http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD329.PDF>

*Did Dijkstra invent everything? No, but maybe close. He certainly was the first to clearly write down what the problems were in concurrent code. However, it is true that practitioners in operating system design knew of many of the problems described by Dijkstra, so perhaps giving him too much credit would be a misrepresentation of history.*

[D08] "The Little Book of Semaphores"

A.B. Downey

Available: <http://greenteapress.com/seahores/>

*A nice (and free!) book about semaphores. Lots of fun problems to solve, if you like that sort of thing.*

[D71] "Hierarchical ordering of sequential processes"

E.W. Dijkstra

Available: <http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF>

*Presents numerous concurrency problems, including the Dining Philosophers. The wikipedia page about this problem is also quite informative.*

[GR92] "Transaction Processing: Concepts and Techniques"

Jim Gray and Andreas Reuter

Morgan Kaufmann, September 1992

*The exact quote that we find particularly humorous is found on page 485, at the top of Section 8.8:*

*"The first multiprocessors, circa 1960, had test and set instructions ... presumably the OS implementors worked out the appropriate algorithms, although Dijkstra is generally credited with inventing semaphores many years later."*

[H87] "Aspects of Cache Memory and Instruction Buffer Performance"

Mark D. Hill

Ph.D. Dissertation, U.C. Berkeley, 1987

*Hill's dissertation work, for those obsessed with caching in early systems. A great example of a quantitative dissertation.*

[L83] "Hints for Computer Systems Design"

Butler Lampson

ACM Operating Systems Review, 15:5, October 1983

*Lampson, a famous systems researcher, loved using hints in the design of computer systems. A hint is something that is often correct but can be wrong; in this use, a signal() is telling a waiting thread that it changed the condition that the waiter was waiting on, but not to trust that the condition will be in the desired state when the waiting thread wakes up. In this paper about hints for designing systems, one of Lampson's general hints is that you should use hints. It is not as confusing as it sounds.*

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [CONFORMING TO](#) |  
[NOTES](#) | [SEE ALSO](#) | [COLOPHON](#)

  
 Search online pages
**FLOCK(2)****Linux Programmer's Manual****FLOCK(2)****NAME**[top](#)

**flock** - apply or remove an advisory lock on an open file

**SYNOPSIS**[top](#)

```
#include <sys/file.h>

int flock(int fd, int operation);
```

**DESCRIPTION**[top](#)

Apply or remove an advisory lock on the open file specified by *fd*. The argument *operation* is one of the following:

**LOCK\_SH** Place a shared lock. More than one process may hold a shared lock for a given file at a given time.

**LOCK\_EX** Place an exclusive lock. Only one process may hold an exclusive lock for a given file at a given time.

**LOCK\_UN** Remove an existing lock held by this process.

A call to **flock()** may block if an incompatible lock is held by another process. To make a nonblocking request, include **LOCK\_NB** (by ORing) with any of the above operations.

A single file may not simultaneously have both shared and exclusive locks.

Locks created by **flock()** are associated with an open file description (see [open\(2\)](#)). This means that duplicate file descriptors (created by, for example, [fork\(2\)](#) or [dup\(2\)](#)) refer to the same lock, and this lock may be modified or released using any of these file descriptors. Furthermore, the lock is released either by an explicit **LOCK\_UN** operation on any of these duplicate file descriptors, or when all such file descriptors have been closed.

If a process uses [open\(2\)](#) (or similar) to obtain more than one file descriptor for the same file, these file descriptors are treated independently by **flock()**. An attempt to lock the file using one of these file descriptors may be denied by a lock that the calling process has already placed via another file descriptor.

A process may hold only one type of lock (shared or exclusive) on a file. Subsequent **flock()** calls on an already locked file will convert an existing lock to the new lock mode.

Locks created by **flock()** are preserved across an **execve(2)**.

A shared or exclusive lock can be placed on a file regardless of the mode in which the file was opened.

## RETURN VALUE

[top](#)

On success, zero is returned. On error, -1 is returned, and **errno** is set appropriately.

## ERRORS

[top](#)

**EBADF** *fd* is not an open file descriptor.

**EINTR** While waiting to acquire a lock, the call was interrupted by delivery of a signal caught by a handler; see **signal(7)**.

**EINVAL** *operation* is invalid.

**ENOLCK** The kernel ran out of memory for allocating lock records.

**EWOULDBLOCK**

The file is locked and the **LOCK\_NB** flag was selected.

## CONFORMING TO

[top](#)

4.4BSD (the **flock()** call first appeared in 4.2BSD). A version of **flock()**, possibly implemented in terms of **fcntl(2)**, appears on most UNIX systems.

## NOTES

[top](#)

Since kernel 2.0, **flock()** is implemented as a system call in its own right rather than being emulated in the GNU C library as a call to **fcntl(2)**. With this implementation, there is no interaction between the types of lock placed by **flock()** and **fcntl(2)**, and **flock()** does not detect deadlock. (Note, however, that on some systems, such as the modern BSDs, **flock()** and **fcntl(2)** locks *do* interact with one another.)

In Linux kernels up to 2.6.11, **flock()** does not lock files over NFS (i.e., the scope of locks was limited to the local system). Instead, one could use **fcntl(2)** byte-range locking, which does work over NFS. given a sufficiently recent version of Linux and a server which supports locking. Since Linux 2.6.12, NFS clients support **flock()**

locks by emulating them as byte-range locks on the entire file. This means that `fcntl(2)` and `flock()` locks *do* interact with one another over NFS. Since Linux 2.6.37, the kernel supports a compatibility mode that allows `flock()` locks (and also `fcntl(2)` byte region locks) to be treated as local; see the discussion of the `local_lock` option in [nfs\(5\)](#).

`flock()` places advisory locks only; given suitable permissions on a file, a process is free to ignore the use of `flock()` and perform I/O on the file.

`flock()` and `fcntl(2)` locks have different semantics with respect to forked processes and `dup(2)`. On systems that implement `flock()` using `fcntl(2)`, the semantics of `flock()` will be different from those described in this manual page.

Converting a lock (shared to exclusive, or vice versa) is not guaranteed to be atomic: the existing lock is first removed, and then a new lock is established. Between these two steps, a pending lock request by another process may be granted, with the result that the conversion either blocks, or fails if `LOCK_NB` was specified. (This is the original BSD behavior, and occurs on many other implementations.)

## SEE ALSO

[top](#)

[flock\(1\)](#), [close\(2\)](#), [dup\(2\)](#), [execve\(2\)](#), [fcntl\(2\)](#), [fork\(2\)](#), [open\(2\)](#), [lockf\(3\)](#), [lslocks\(8\)](#)

[Documentation/filesystems/locks.txt](#) in the Linux kernel source tree ([Documentation/locks.txt](#) in older kernels)

## COLOPHON

[top](#)

This page is part of release 4.08 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at <https://www.kernel.org/doc/man-pages/>.

Linux

2016-03-15

FLOCK(2)

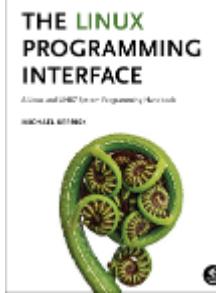
[Copyright and license for this manual page](#)

HTML rendering created 2016-10-08 by Michael Kerrisk, author of *The Linux Programming Interface*, maintainer of the Linux *man-pages* project.

For details of in-depth Linux/UNIX system programming training courses that I teach, look [here](#).

Hosting by [jambit GmbH](#).





[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [ATTRIBUTES](#) |  
[CONFORMING TO](#) | [SEE ALSO](#) | [COLOPHON](#)

  
 Search online pages
**LOCKF(3)****Linux Programmer's Manual****LOCKF(3)****NAME**[top](#)

`lockf` - apply, test or remove a POSIX lock on an open file

**SYNOPSIS**[top](#)

```
#include <unistd.h>

int lockf(int fd, int cmd, off_t len);
```

Feature Test Macro Requirements for glibc (see [feature\\_test\\_macros\(7\)](#)):

```
lockf():
    _XOPEN_SOURCE >= 500
    || /* Glibc since 2.19: */ _DEFAULT_SOURCE
    || /* Glibc versions <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE
```

**DESCRIPTION**[top](#)

Apply, test or remove a POSIX lock on a section of an open file. The file is specified by *fd*, a file descriptor open for writing, the action by *cmd*, and the section consists of byte positions *pos..pos+len-1* if *len* is positive, and *pos-len..pos-1* if *len* is negative, where *pos* is the current file position, and if *len* is zero, the section extends from the current file position to infinity, encompassing the present and future end-of-file positions. In all cases, the section may extend past current end-of-file.

On Linux, `lockf()` is just an interface on top of `fcntl(2)` locking. Many other systems implement `lockf()` in this way, but note that POSIX.1 leaves the relationship between `lockf()` and `fcntl(2)` locks unspecified. A portable application should probably avoid mixing calls to these interfaces.

Valid operations are given below:

**F\_LOCK** Set an exclusive lock on the specified section of the file. If (part of) this section is already locked, the call blocks until the previous lock is released. If this section overlaps an earlier locked section, both are merged. File locks are released as soon as the process holding the locks closes some file descriptor for the file. A child process does not inherit these locks.

**F\_TLOCK**

Same as **F\_LOCK** but the call never blocks and returns an error instead if the file is already locked.

**F\_ULOCK**

Unlock the indicated section of the file. This may cause a locked section to be split into two locked sections.

**F\_TEST** Test the lock: return 0 if the specified section is unlocked or locked by this process; return -1, set *errno* to **EAGAIN** (**EACCES** on some other systems), if another process holds a lock.

**RETURN VALUE**[top](#)

On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

**ERRORS**[top](#)**EACCES** or **EAGAIN**

The file is locked and **F\_TLOCK** or **F\_TEST** was specified, or the operation is prohibited because the file has been memory-mapped by another process.

**EBADF** *fd* is not an open file descriptor; or *cmd* is **F\_LOCK** or **F\_TLOCK** and *fd* is not a writable file descriptor.

**EDEADLK**

The command was **F\_LOCK** and this lock operation would cause a deadlock.

**EINVAL** An invalid operation was specified in *cmd*.

**ENOLCK** Too many segment locks open, lock table is full.

**ATTRIBUTES**[top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

Interface	Attribute	Value
<b>lockf()</b>	Thread safety	MT-Safe

**CONFORMING TO**[top](#)

**SEE ALSO**[top](#)[fcntl\(2\)](#), [flock\(2\)](#)

*locks.txt* and *mandatory-locking.txt* in the Linux kernel source directory *Documentation/filesystems* (on older kernels, these files are directly under the *Documentation* directory, and *mandatory-locking.txt* is called *mandatory.txt*)

**COLOPHON**[top](#)

This page is part of release 4.08 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at  
<https://www.kernel.org/doc/man-pages/>.

**GNU****2016-03-15****LOCKF(3)**

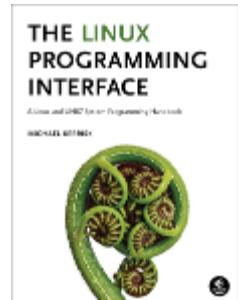
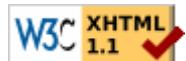
---

[Copyright and license for this manual page](#)

HTML rendering created 2016-10-08 by Michael Kerrisk, author of *The Linux Programming Interface*, maintainer of the Linux *man-pages* project.

For details of in-depth Linux/UNIX system programming training courses that I teach, look [here](#).

Hosting by [jambit GmbH](#).



## Common Concurrency Problems

Researchers have spent a great deal of time and effort looking into concurrency bugs over many years. Much of the early work focused on **deadlock**, a topic which we've touched on in the past chapters but will now dive into deeply [C+71]. More recent work focuses on studying other types of common concurrency bugs (i.e., non-deadlock bugs). In this chapter, we take a brief look at some example concurrency problems found in real code bases, to better understand what problems to look out for. And thus our central issue for this chapter:

### CRUX: HOW TO HANDLE COMMON CONCURRENCY BUGS

Concurrency bugs tend to come in a variety of common patterns. Knowing which ones to look out for is the first step to writing more robust, correct concurrent code.

### 32.1 What Types Of Bugs Exist?

The first, and most obvious, question is this: what types of concurrency bugs manifest in complex, concurrent programs? This question is difficult to answer in general, but fortunately, some others have done the work for us. Specifically, we rely upon a study by Lu et al. [L+08], which analyzes a number of popular concurrent applications in great detail to understand what types of bugs arise in practice.

The study focuses on four major and important open-source applications: MySQL (a popular database management system), Apache (a well-known web server), Mozilla (the famous web browser), and OpenOffice (a free version of the MS Office suite, which some people actually use). In the study, the authors examine concurrency bugs that have been found and fixed in each of these code bases, turning the developers' work into a quantitative bug analysis; understanding these results can help you understand what types of problems actually occur in mature code bases.

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
OpenOffice	Office Suite	6	2
Total		74	31

Figure 32.1: Bugs In Modern Applications

Figure 32.1 shows a summary of the bugs Lu and colleagues studied. From the figure, you can see that there were 105 total bugs, most of which were not deadlock (74); the remaining 31 were deadlock bugs. Further, you can see that the number of bugs studied from each application; while OpenOffice only had 8 total concurrency bugs, Mozilla had nearly 60.

We now dive into these different classes of bugs (non-deadlock, deadlock) a bit more deeply. For the first class of non-deadlock bugs, we use examples from the study to drive our discussion. For the second class of deadlock bugs, we discuss the long line of work that has been done in either preventing, avoiding, or handling deadlock.

## 32.2 Non-Deadlock Bugs

Non-deadlock bugs make up a majority of concurrency bugs, according to Lu's study. But what types of bugs are these? How do they arise? How can we fix them? We now discuss the two major types of non-deadlock bugs found by Lu et al.: **atomicity violation** bugs and **order violation** bugs.

### Atomicity-Violation Bugs

The first type of problem encountered is referred to as an **atomicity violation**. Here is a simple example, found in MySQL. Before reading the explanation, try figuring out what the bug is. Do it!

```

1 Thread 1::
2 if (thd->proc_info) {
3     ...
4     fputs(thd->proc_info, ...);
5     ...
6 }
7
8 Thread 2::
9 thd->proc_info = NULL;

```

In the example, two different threads access the field `proc_info` in the structure `thd`. The first thread checks if the value is non-NULL and then prints its value; the second thread sets it to NULL. Clearly, if the first thread performs the check but then is interrupted before the call to `fputs`, the second thread could run in-between, thus setting the pointer to NULL; when the first thread resumes, it will crash, as a NULL pointer will be dereferenced by `fputs`.

The more formal definition of an atomicity violation, according to Lu et al, is this: “The desired serializability among multiple memory accesses is violated (i.e. a code region is intended to be atomic, but the atomicity is not enforced during execution).” In our example above, the code has an *atomicity assumption* (in Lu’s words) about the check for non-NULL of `proc_info` and the usage of `proc_info` in the `fputs()` call; when the assumption is incorrect, the code will not work as desired.

Finding a fix for this type of problem is often (but not always) straightforward. Can you think of how to fix the code above?

In this solution, we simply add locks around the shared-variable references, ensuring that when either thread accesses the `proc_info` field, it has a lock held (`proc_info_lock`). Of course, any other code that accesses the structure should also acquire this lock before doing so.

```

1 pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;
2
3 Thread 1::
4 pthread_mutex_lock(&proc_info_lock);
5 if (thd->proc_info) {
6     ...
7     fputs(thd->proc_info, ...);
8     ...
9 }
10 pthread_mutex_unlock(&proc_info_lock);
11
12 Thread 2::
13 pthread_mutex_lock(&proc_info_lock);
14 thd->proc_info = NULL;
15 pthread_mutex_unlock(&proc_info_lock);

```

## Order-Violation Bugs

Another common type of non-deadlock bug found by Lu et al. is known as an **order violation**. Here is another simple example; once again, see if you can figure out why the code below has a bug in it.

```

1 Thread 1::
2 void init() {
3     ...
4     mThread = PR_CreateThread(mMain, ...);
5     ...
6 }
7
8 Thread 2::
9 void mMain(...) {
10     ...
11     mState = mThread->State;
12     ...
13 }

```

As you probably figured out, the code in Thread 2 seems to assume that the variable `mThread` has already been initialized (and is not NULL); however, if Thread 2 runs immediately once created, the value of `mThread` will not be set when it is accessed within `mMain()` in Thread 2, and will

likely crash with a NULL-pointer dereference. Note that we assume the value of `mThread` is initially NULL; if not, even stranger things could happen as arbitrary memory locations are accessed through the dereference in Thread 2.

The more formal definition of an order violation is this: “The desired order between two (groups of) memory accesses is flipped (i.e., *A* should always be executed before *B*, but the order is not enforced during execution)” [L+08].

The fix to this type of bug is generally to enforce ordering. As we discussed in detail previously, using **condition variables** is an easy and robust way to add this style of synchronization into modern code bases. In the example above, we could thus rewrite the code as follows:

```

1  pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
3  int mtInit           = 0;
4
5  Thread 1::
6  void init() {
7      ...
8      mThread = PR_CreateThread(mMain, ...);
9
10     // signal that the thread has been created...
11     pthread_mutex_lock(&mtLock);
12     mtInit = 1;
13     pthread_cond_signal(&mtCond);
14     pthread_mutex_unlock(&mtLock);
15     ...
16 }
17
18 Thread 2::
19 void mMain(...) {
20     ...
21     // wait for the thread to be initialized...
22     pthread_mutex_lock(&mtLock);
23     while (mtInit == 0)
24         pthread_cond_wait(&mtCond, &mtLock);
25     pthread_mutex_unlock(&mtLock);
26
27     mState = mThread->State;
28     ...
29 }
```

In this fixed-up code sequence, we have added a lock (`mtLock`) and corresponding condition variable (`mtCond`), as well as a state variable (`mtInit`). When the initialization code runs, it sets the state of `mtInit` to 1 and signals that it has done so. If Thread 2 had run before this point, it will be waiting for this signal and corresponding state change; if it runs later, it will check the state and see that the initialization has already occurred (i.e., `mtInit` is set to 1), and thus continue as is proper. Note that we could likely use `mThread` as the state variable itself, but do not do so for the sake of simplicity here. When ordering matters between threads, condition variables (or semaphores) can come to the rescue.

## Non-Deadlock Bugs: Summary

A large fraction (97%) of non-deadlock bugs studied by Lu et al. are either atomicity or order violations. Thus, by carefully thinking about these types of bug patterns, programmers can likely do a better job of avoiding them. Moreover, as more automated code-checking tools develop, they should likely focus on these two types of bugs as they constitute such a large fraction of non-deadlock bugs found in deployment.

Unfortunately, not all bugs are as easily fixable as the examples we looked at above. Some require a deeper understanding of what the program is doing, or a larger amount of code or data structure reorganization to fix. Read Lu et al.'s excellent (and readable) paper for more details.

### 32.3 Deadlock Bugs

Beyond the concurrency bugs mentioned above, a classic problem that arises in many concurrent systems with complex locking protocols is known as **deadlock**. Deadlock occurs, for example, when a thread (say Thread 1) is holding a lock (`L1`) and waiting for another one (`L2`); unfortunately, the thread (Thread 2) that holds lock `L2` is waiting for `L1` to be released. Here is a code snippet that demonstrates such a potential deadlock:

```
Thread 1:           Thread 2:  
pthread_mutex_lock(L1);   pthread_mutex_lock(L2);  
pthread_mutex_lock(L2);   pthread_mutex_lock(L1);
```

Note that if this code runs, deadlock does not necessarily occur; rather, it may occur, if, for example, Thread 1 grabs lock `L1` and then a context switch occurs to Thread 2. At that point, Thread 2 grabs `L2`, and tries to acquire `L1`. Thus we have a deadlock, as each thread is waiting for the other and neither can run. See Figure 32.2 for a graphical depiction; the presence of a **cycle** in the graph is indicative of the deadlock.

The figure should make clear the problem. How should programmers write code so as to handle deadlock in some way?

#### CRUX: HOW TO DEAL WITH DEADLOCK

How should we build systems to prevent, avoid, or at least detect and recover from deadlock? Is this a real problem in systems today?

### Why Do Deadlocks Occur?

As you may be thinking, simple deadlocks such as the one above seem readily avoidable. For example, if Thread 1 and 2 both made sure to grab locks in the same order, the deadlock would never arise. So why do deadlocks happen?

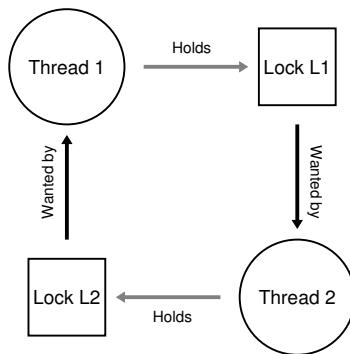


Figure 32.2: The Deadlock Dependency Graph

One reason is that in large code bases, complex dependencies arise between components. Take the operating system, for example. The virtual memory system might need to access the file system in order to page in a block from disk; the file system might subsequently require a page of memory to read the block into and thus contact the virtual memory system. Thus, the design of locking strategies in large systems must be carefully done to avoid deadlock in the case of circular dependencies that may occur naturally in the code.

Another reason is due to the nature of **encapsulation**. As software developers, we are taught to hide details of implementations and thus make software easier to build in a modular way. Unfortunately, such modularity does not mesh well with locking. As Jula et al. point out [J+08], some seemingly innocuous interfaces almost invite you to deadlock. For example, take the Java Vector class and the method `AddAll()`. This routine would be called as follows:

```
Vector v1, v2;
v1.AddAll(v2);
```

Internally, because the method needs to be multi-thread safe, locks for both the vector being added to (`v1`) and the parameter (`v2`) need to be acquired. The routine acquires said locks in some arbitrary order (say `v1` then `v2`) in order to add the contents of `v2` to `v1`. If some other thread calls `v2.AddAll(v1)` at nearly the same time, we have the potential for deadlock, all in a way that is quite hidden from the calling application.

## Conditions for Deadlock

Four conditions need to hold for a deadlock to occur [C+71]:

- **Mutual exclusion:** Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).
- **Hold-and-wait:** Threads hold resources allocated to them (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire).
- **No preemption:** Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.
- **Circular wait:** There exists a circular chain of threads such that each thread holds one or more resources (e.g., locks) that are being requested by the next thread in the chain.

If any of these four conditions are not met, deadlock cannot occur. Thus, we first explore techniques to *prevent* deadlock; each of these strategies seeks to prevent one of the above conditions from arising and thus is one approach to handling the deadlock problem.

## Prevention

### Circular Wait

Probably the most practical prevention technique (and certainly one that is frequently employed) is to write your locking code such that you never induce a circular wait. The most straightforward way to do that is to provide a **total ordering** on lock acquisition. For example, if there are only two locks in the system (`L1` and `L2`), you can prevent deadlock by always acquiring `L1` before `L2`. Such strict ordering ensures that no cyclical wait arises; hence, no deadlock.

Of course, in more complex systems, more than two locks will exist, and thus total lock ordering may be difficult to achieve (and perhaps is unnecessary anyhow). Thus, a **partial ordering** can be a useful way to structure lock acquisition so as to avoid deadlock. An excellent real example of partial lock ordering can be seen in the memory mapping code in Linux [T+94]; the comment at the top of the source code reveals ten different groups of lock acquisition orders, including simple ones such as “`i_mutex before i_mmap_mutex`” and more complex orders such as “`i_mmap_mutex before private_lock before swap_lock before mapping->tree_lock`”.

As you can imagine, both total and partial ordering require careful design of locking strategies and must be constructed with great care. Further, ordering is just a convention, and a sloppy programmer can easily ignore the locking protocol and potentially cause deadlock. Finally, lock

**TIP: ENFORCE LOCK ORDERING BY LOCK ADDRESS**

In some cases, a function must grab two (or more) locks; thus, we know we must be careful or deadlock could arise. Imagine a function that is called as follows: `do_something(mutex_t *m1, mutex_t *m2)`. If the code always grabs `m1` before `m2` (or always `m2` before `m1`), it could deadlock, because one thread could call `do_something(L1, L2)` while another thread could call `do_something(L2, L1)`.

To avoid this particular issue, the clever programmer can use the *address* of each lock as a way of ordering lock acquisition. By acquiring locks in either high-to-low or low-to-high address order, `do_something()` can guarantee that it always acquires locks in the same order, regardless of which order they are passed in. The code would look something like this:

```
if (m1 > m2) { // grab locks in high-to-low address order
    pthread_mutex_lock(m1);
    pthread_mutex_lock(m2);
} else {
    pthread_mutex_lock(m2);
    pthread_mutex_lock(m1);
}
// Code assumes that m1 != m2 (it is not the same lock)
```

By using this simple technique, a programmer can ensure a simple and efficient deadlock-free implementation of multi-lock acquisition.

ordering requires a deep understanding of the code base, and how various routines are called; just one mistake could result in the “D” word<sup>1</sup>.

**Hold-and-wait**

The hold-and-wait requirement for deadlock can be avoided by acquiring all locks at once, atomically. In practice, this could be achieved as follows:

```
1   pthread_mutex_lock(prevention); // begin lock acquisition
2   pthread_mutex_lock(L1);
3   pthread_mutex_lock(L2);
4   ...
5   pthread_mutex_unlock(prevention); // end
```

By first grabbing the lock `prevention`, this code guarantees that no untimely thread switch can occur in the midst of lock acquisition and thus deadlock can once again be avoided. Of course, it requires that any time any thread grabs a lock, it first acquires the global prevention lock. For example, if another thread was trying to grab locks `L1` and `L2` in a different order, it would be OK, because it would be holding the prevention lock while doing so.

---

<sup>1</sup>Hint: “D” stands for “Deadlock”.

Note that the solution is problematic for a number of reasons. As before, encapsulation works against us: when calling a routine, this approach requires us to know exactly which locks must be held and to acquire them ahead of time. This technique also is likely to decrease concurrency as all locks must be acquired early on (at once) instead of when they are truly needed.

## No Preemption

Because we generally view locks as held until unlock is called, multiple lock acquisition often gets us into trouble because when waiting for one lock we are holding another. Many thread libraries provide a more flexible set of interfaces to help avoid this situation. Specifically, the routine `pthread_mutex_trylock()` either grabs the lock (if it is available) and returns success or returns an error code indicating the lock is held; in the latter case, you can try again later if you want to grab that lock.

Such an interface could be used as follows to build a deadlock-free, ordering-robust lock acquisition protocol:

```
1 top:  
2     pthread_mutex_lock(L1);  
3     if (pthread_mutex_trylock(L2) != 0) {  
4         pthread_mutex_unlock(L1);  
5         goto top;  
6     }
```

Note that another thread could follow the same protocol but grab the locks in the other order (`L2` then `L1`) and the program would still be deadlock free. One new problem does arise, however: **livelock**. It is possible (though perhaps unlikely) that two threads could both be repeatedly attempting this sequence and repeatedly failing to acquire both locks. In this case, both systems are running through this code sequence over and over again (and thus it is not a deadlock), but progress is not being made, hence the name livelock. There are solutions to the livelock problem, too: for example, one could add a random delay before looping back and trying the entire thing over again, thus decreasing the odds of repeated interference among competing threads.

One final point about this solution: it skirts around the hard parts of using a trylock approach. The first problem that would likely exist again arises due to encapsulation: if one of these locks is buried in some routine that is getting called, the jump back to the beginning becomes more complex to implement. If the code had acquired some resources (other than `L1`) along the way, it must make sure to carefully release them as well; for example, if after acquiring `L1`, the code had allocated some memory, it would have to release that memory upon failure to acquire `L2`, before jumping back to the top to try the entire sequence again. However, in limited circumstances (e.g., the Java vector method mentioned earlier), this type of approach could work well.

## Mutual Exclusion

The final prevention technique would be to avoid the need for mutual exclusion at all. In general, we know this is difficult, because the code we wish to run does indeed have critical sections. So what can we do?

Herlihy had the idea that one could design various data structures without locks at all [H91, H93]. The idea behind these **lock-free** (and related **wait-free**) approaches here is simple: using powerful hardware instructions, you can build data structures in a manner that does not require explicit locking.

As a simple example, let us assume we have a compare-and-swap instruction, which as you may recall is an atomic instruction provided by the hardware that does the following:

```

1 int CompareAndSwap(int *address, int expected, int new) {
2     if (*address == expected) {
3         *address = new;
4         return 1; // success
5     }
6     return 0; // failure
7 }
```

Imagine we now wanted to atomically increment a value by a certain amount. We could do it as follows:

```

1 void AtomicIncrement(int *value, int amount) {
2     do {
3         int old = *value;
4     } while (CompareAndSwap(value, old, old + amount) == 0);
5 }
```

Instead of acquiring a lock, doing the update, and then releasing it, we have instead built an approach that repeatedly tries to update the value to the new amount and uses the compare-and-swap to do so. In this manner, no lock is acquired, and no deadlock can arise (though livelock is still a possibility).

Let us consider a slightly more complex example: list insertion. Here is code that inserts at the head of a list:

```

1 void insert(int value) {
2     node_t *n = malloc(sizeof(node_t));
3     assert(n != NULL);
4     n->value = value;
5     n->next = head;
6     head = n;
7 }
```

This code performs a simple insertion, but if called by multiple threads at the “same time”, has a race condition (see if you can figure out why). Of course, we could solve this by surrounding this code with a lock acquire and release:

```

1 void insert(int value) {
2     node_t *n = malloc(sizeof(node_t));
3     assert(n != NULL);
4     n->value = value;
5     pthread_mutex_lock(listlock);    // begin critical section
6     n->next = head;
7     head = n;
8     pthread_mutex_unlock(listlock); // end critical section
9 }
```

In this solution, we are using locks in the traditional manner<sup>2</sup>. Instead, let us try to perform this insertion in a lock-free manner simply using the compare-and-swap instruction. Here is one possible approach:

```

1 void insert(int value) {
2     node_t *n = malloc(sizeof(node_t));
3     assert(n != NULL);
4     n->value = value;
5     do {
6         n->next = head;
7     } while (!CompareAndSwap(&head, n->next, n) == 0);
8 }
```

The code here updates the next pointer to point to the current head, and then tries to swap the newly-created node into position as the new head of the list. However, this will fail if some other thread successfully swapped in a new head in the meanwhile, causing this thread to retry again with the new head.

Of course, building a useful list requires more than just a list insert, and not surprisingly building a list that you can insert into, delete from, and perform lookups on in a lock-free manner is non-trivial. Read the rich literature on lock-free and wait-free synchronization to learn more [H01, H91, H93].

## Deadlock Avoidance via Scheduling

Instead of deadlock prevention, in some scenarios deadlock **avoidance** is preferable. Avoidance requires some global knowledge of which locks various threads might grab during their execution, and subsequently schedules said threads in a way as to guarantee no deadlock can occur.

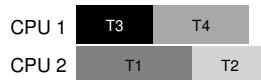
For example, assume we have two processors and four threads which must be scheduled upon them. Assume further we know that Thread 1 (T1) grabs locks L1 and L2 (in some order, at some point during its execution), T2 grabs L1 and L2 as well, T3 grabs just L2, and T4 grabs no locks at all. We can show these lock acquisition demands of the threads in tabular form:

---

<sup>2</sup>The astute reader might be asking why we grabbed the lock so late, instead of right when entering `insert()`; can you, astute reader, figure out why that is likely correct? What assumptions does the code make, for example, about the call to `malloc()`?

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

A smart scheduler could thus compute that as long as T1 and T2 are not run at the same time, no deadlock could ever arise. Here is one such schedule:



Note that it is OK for (T3 and T1) or (T3 and T2) to overlap. Even though T3 grabs lock L2, it can never cause a deadlock by running concurrently with other threads because it only grabs one lock.

Let's look at one more example. In this one, there is more contention for the same resources (again, locks L1 and L2), as indicated by the following contention table:

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

In particular, threads T1, T2, and T3 all need to grab both locks L1 and L2 at some point during their execution. Here is a possible schedule that guarantees that no deadlock could ever occur:



As you can see, static scheduling leads to a conservative approach where T1, T2, and T3 are all run on the same processor, and thus the total time to complete the jobs is lengthened considerably. Though it may have been possible to run these tasks concurrently, the fear of deadlock prevents us from doing so, and the cost is performance.

One famous example of an approach like this is Dijkstra's Banker's Algorithm [D64], and many similar approaches have been described in the literature. Unfortunately, they are only useful in very limited environments, for example, in an embedded system where one has full knowledge of the entire set of tasks that must be run and the locks that they need. Further, such approaches can limit concurrency, as we saw in the second example above. Thus, avoidance of deadlock via scheduling is not a widely-used general-purpose solution.

## Detect and Recover

One final general strategy is to allow deadlocks to occasionally occur, and then take some action once such a deadlock has been detected. For example, if an OS froze once a year, you would just reboot it and get happily (or

**TIP: DON'T ALWAYS DO IT PERFECTLY (TOM WEST'S LAW)**

Tom West, famous as the subject of the classic computer-industry book *Soul of a New Machine* [K81], says famously: "Not everything worth doing is worth doing well", which is a terrific engineering maxim. If a bad thing happens rarely, certainly one should not spend a great deal of effort to prevent it, particularly if the cost of the bad thing occurring is small. If, on the other hand, you are building a space shuttle, and the cost of something going wrong is the space shuttle blowing up, well, perhaps you should ignore this piece of advice.

grumpily) on with your work. If deadlocks are rare, such a non-solution is indeed quite pragmatic.

Many database systems employ deadlock detection and recovery techniques. A deadlock detector runs periodically, building a resource graph and checking it for cycles. In the event of a cycle (deadlock), the system needs to be restarted. If more intricate repair of data structures is first required, a human being may be involved to ease the process.

More detail on database concurrency, deadlock, and related issues can be found elsewhere [B+87, K87]. Read these works, or better yet, take a course on databases to learn more about this rich and interesting topic.

## 32.4 Summary

In this chapter, we have studied the types of bugs that occur in concurrent programs. The first type, non-deadlock bugs, are surprisingly common, but often are easier to fix. They include atomicity violations, in which a sequence of instructions that should have been executed together was not, and order violations, in which the needed order between two threads was not enforced.

We have also briefly discussed deadlock: why it occurs, and what can be done about it. The problem is as old as concurrency itself, and many hundreds of papers have been written about the topic. The best solution in practice is to be careful, develop a lock acquisition order, and thus prevent deadlock from occurring in the first place. Wait-free approaches also have promise, as some wait-free data structures are now finding their way into commonly-used libraries and critical systems, including Linux. However, their lack of generality and the complexity to develop a new wait-free data structure will likely limit the overall utility of this approach. Perhaps the best solution is to develop new concurrent programming models: in systems such as MapReduce (from Google) [GD02], programmers can describe certain types of parallel computations without any locks whatsoever. Locks are problematic by their very nature; perhaps we should seek to avoid using them unless we truly must.

## References

- [B+87] "Concurrency Control and Recovery in Database Systems"  
 Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman  
 Addison-Wesley, 1987  
*The classic text on concurrency in database management systems. As you can tell, understanding concurrency, deadlock, and other topics in the world of databases is a world unto itself. Study it and find out for yourself.*
- [C+71] "System Deadlocks"  
 E.G. Coffman, M.J. Elphick, A. Shoshani  
 ACM Computing Surveys, 3:2, June 1971  
*The classic paper outlining the conditions for deadlock and how you might go about dealing with it. There are certainly some earlier papers on this topic; see the references within this paper for details.*
- [D64] "Een algorithme ter voorkoming van de dodelijke omarming"  
 Edsger Dijkstra  
 Circulated privately, around 1964  
 Available: <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF>  
*Indeed, not only did Dijkstra come up with a number of solutions to the deadlock problem, he was the first to note its existence, at least in written form. However, he called it the "deadly embrace", which (thankfully) did not catch on.*
- [GD02] "MapReduce: Simplified Data Processing on Large Clusters"  
 Sanjay Ghemawat and Jeff Dean  
 OSDI '04, San Francisco, CA, October 2004  
*The MapReduce paper ushered in the era of large-scale data processing, and proposes a framework for performing such computations on clusters of generally unreliable machines.*
- [H01] "A Pragmatic Implementation of Non-blocking Linked-lists"  
 Tim Harris  
 International Conference on Distributed Computing (DISC), 2001  
*A relatively modern example of the difficulties of building something as simple as a concurrent linked list without locks.*
- [H91] "Wait-free Synchronization"  
 Maurice Herlihy  
 ACM TOPLAS, 13:1, January 1991  
*Herlihy's work pioneers the ideas behind wait-free approaches to writing concurrent programs. These approaches tend to be complex and hard, often more difficult than using locks correctly, probably limiting their success in the real world.*
- [H93] "A Methodology for Implementing Highly Concurrent Data Objects"  
 Maurice Herlihy  
 ACM TOPLAS, 15:5, November 1993  
*A nice overview of lock-free and wait-free structures. Both approaches eschew locks, but wait-free approaches are harder to realize, as they try to ensure that any operation on a concurrent structure will terminate in a finite number of steps (e.g., no unbounded looping).*
- [J+08] "Deadlock Immunity: Enabling Systems To Defend Against Deadlocks"  
 Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, George Canea  
 OSDI '08, San Diego, CA, December 2008  
*An excellent recent paper on deadlocks and how to avoid getting caught in the same ones over and over again in a particular system.*

[K81] "Soul of a New Machine"

Tracy Kidder, 1980

*A must-read for any systems builder or engineer, detailing the early days of how a team inside Data General (DG), led by Tom West, worked to produce a "new machine." Kidder's other books are also excellent, including Mountains beyond Mountains. Or maybe you don't agree with us, comma?*

[K87] "Deadlock Detection in Distributed Databases"

Edgar Knapp

ACM Computing Surveys, 19:4, December 1987

*An excellent overview of deadlock detection in distributed database systems. Also points to a number of other related works, and thus is a good place to start your reading.*

[L+08] "Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics"

Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou

ASPLOS '08, March 2008, Seattle, Washington

*The first in-depth study of concurrency bugs in real software, and the basis for this chapter. Look at Y.Y. Zhou's or Shan Lu's web pages for many more interesting papers on bugs.*

[T+94] "Linux File Memory Map Code"

Linus Torvalds and many others

Available: <http://lxr.free-electrons.com/source/mm/filemap.c>

*Thanks to Michael Waltrip (NYU) for pointing out this precious example. The real world, as you can see in this file, can be a bit more complex than the simple clarity found in textbooks...*

## Homework

This homework lets you explore some real code that deadlocks (or avoids deadlock). The different versions of code correspond to different approaches to avoiding deadlock in a simplified `vector_add()` routine. Specifically:

- `vector-deadlock.c`: This version of `vector_add()` does not try to avoid deadlock and thus may indeed do so.
- `vector-global-order.c`: This version acquires locks in a global order to avoid deadlock.
- `vector-try-wait.c`: This version is willing to release a lock when it senses deadlock might occur.
- `vector-avoid-hold-and-wait.c`: This version uses a global lock around lock acquisition to avoid deadlock.
- `vector-nolock.c`: This version uses an atomic fetch-and-add instead of locks.

See the README for details on these programs and their common substrate.

## Questions

1. First let's make sure you understand how the programs generally work, and some of the key options. Study the code in the file called `vector-deadlock.c`, as well as in `main-common.c` and related files.

Now, run `./vector-deadlock -n 2 -l 1 -v`, which instantiates two threads (`-n 2`), each of which does one vector add (`-l 1`), and does so in verbose mode (`-v`). Make sure you understand the output. How does the output change from run to run?

2. Now add the `-d` flag, and change the number of loops (`-l`) from 1 to higher numbers. What happens? Does the code (always) deadlock?
3. How does changing the number of threads (`-n`) change the outcome of the program? Are there any values of `-n` that ensure no deadlock occurs?
4. Now examine the code in `vector-global-order.c`. First, make sure you understand what the code is trying to do; do you understand why the code avoids deadlock? Also, why is there a special case in this `vector_add()` routine when the source and destination vectors are the same?

5. Now run the code with the following flags: `-t -n 2 -l 100000 -d`. How long does the code take to complete? How does the total time change when you increase the number of loops, or the number of threads?
6. What happens if you turn on the parallelism flag (`-p`)? How much would you expect performance to change when each thread is working on adding different vectors (which is what `-p` enables) versus working on the same ones?
7. Now let's study `vector-try-wait.c`. First make sure you understand the code. Is the first call to `pthread_mutex_trylock()` really needed?  
Now run the code. How fast does it run compared to the global order approach? How does the number of retries, as counted by the code, change as the number of threads increases?
8. Now let's look at `vector-avoid-hold-and-wait.c`. What is the main problem with this approach? How does its performance compare to the other versions, when running both with `-p` and without it?
9. Finally, let's look at `vector-nolock.c`. This version doesn't use locks at all; does it provide the exact same semantics as the other versions? Why or why not?
10. Now compare its performance to the other versions, both when threads are working on the same two vectors (no `-p`) and when each thread is working on separate vectors (`-p`). How does this no-lock version perform?

# An Introduction to DOS FAT Volume and File Structure

Mark Kampe [markk@cs.ucla.edu](mailto:markk@cs.ucla.edu)

## 1. Introduction

When the first personal computers with disks became available, they were very small (a few megabytes of disk and a few dozen kilobytes of memory). A file system implementation for such machines had to impose very little overhead on disk space, and be small enough to fit in the BIOS ROM. BIOS stands for **BASIC I/O Subsystem**. Note that the first word is all upper-case. The purpose of the BIOS ROM was to provide run-time support for a BASIC interpreter (which is what Bill Gates did for a living before building DOS). DOS was never intended to provide the features and performance of real timesharing systems.

Disk and memory size have increased in the last thirty years, People now demand state-of-the-art power and functionality from their PCs. Despite the evolution that the last decades have seen, old standards die hard. Much as European train tracks maintain the same wheel spacing used by Roman chariots, most modern OSs still support DOS FAT file systems. DOS file systems are not merely around for legacy reasons. The ISO 9660 CDROM file system format is a descendent of the DOS file system.

The DOS FAT file system is worth studying because:

- It is heavily used all over the world, and is the basis for more modern file system (like 9660).
- It provides reasonable performance (large transfers and well clustered allocation) with a very simple implementation.
- It is a very successful example of "linked list" space allocation.

## 2. Structural Overview

All file systems include a few basic types of data structures:

- bootstrap

code to be loaded into memory and executed when the computer is powered on. MVS volumes reserve the entire first track of the first cylinder for the boot strap.

- volume descriptors

information describing the size, type, and layout of the file system ... and in particular how to find the other key meta-data descriptors.

- file descriptors

information that describes a file (ownership, protection, time of last update, etc.) and points where the actual data is stored on the disk.

- free space descriptors

lists of blocks of (currently) unused space that can be allocated to files.

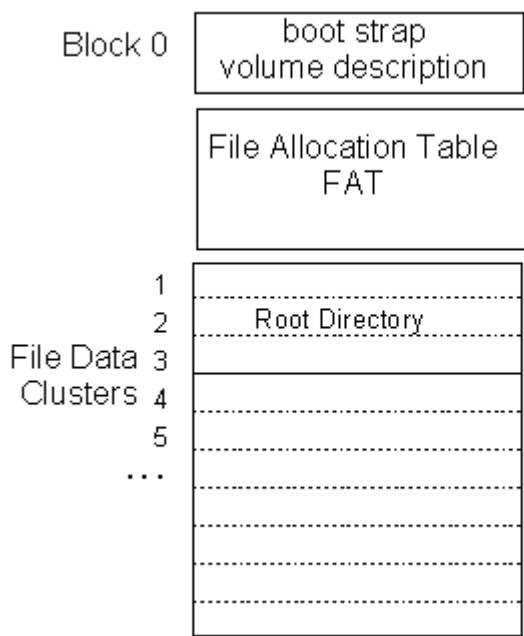
- file name descriptors

data structures that user-chosen names with each file.

DOS FAT file systems divide the volume into fixed-sized (physical) blocks, which are grouped into larger fixed-sized (logical) block clusters.

The first block of DOS FAT volume contains the bootstrap, along with some volume description information. After this comes a much longer **File Allocation Table** (FAT from which the file system takes its name). The File Allocation Table is used, both as a free list, and to keep track of which blocks have been allocated to which files.

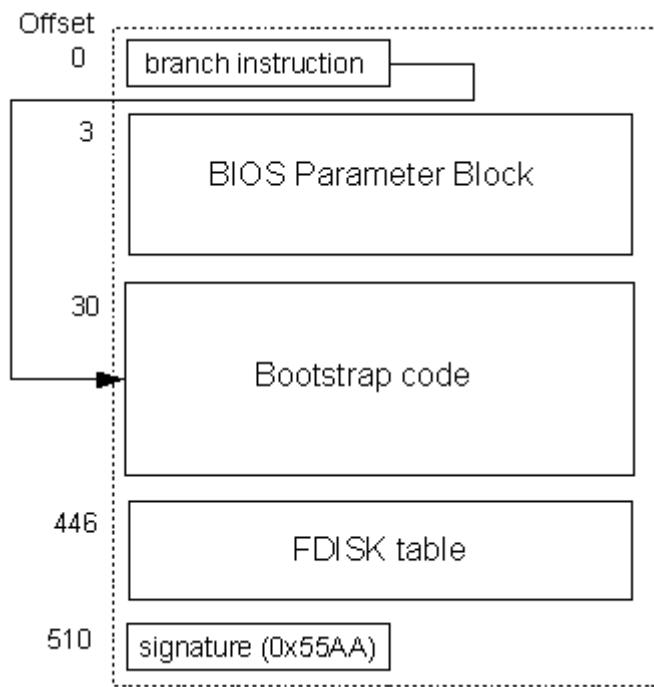
The remainder of the volume is data clusters, which can be allocated to files and directories. The first file on the volume is the root directory, the top of the tree from which all other files and directories on the volume can be reached.



### 3. Boot block BIOS Parameter Block and FDISK Table

Most file systems separate the first block (pure bootstrap code) from volume description information. DOS file systems often combine these into a single block. The format varies between (partitioned) hard disks and (unpartitioned) floppies, and between various releases of DOS and Windows ... but conceptually, the boot record:

- begins with a branch instruction (to the start of the real bootstrap code).
- followed by a volume description (BIOS Parameter Block)
- followed by the real bootstrap code
- followed by an optional disk partitioning table
- followed by a signature (for error checking).



### 3.1 BIOS Parameter Block

After the first few bytes of the bootstrap comes the BIOS parameter block, which contains a brief summary of the device and file system. It describes the device geometry:

- number of bytes per (physical) sector
- number of sectors per track
- number of tracks per cylinder
- total number of sectors on the volume

It also describes the way the file system is layed out on the volume:

- number of sectors per (logical) cluster
- the number of reserved sectors (not part of file system)
- the number of Alternate File Allocation Tables
- the number of entries in the root directory

These parameters enable the OS to interpret the remainder of the file system.

### 3.2 FDISK Table

As disks got larger, the people at MicroSoft figured out that their customers might want to put multiple file systems on each disk. This meant they needed some way of partitioning the disk into logical sub-disks. To do this, they added a small partition table (sometimes called the FDISK table, because of the program that managed it) to the end of the boot strap block.

This FDISK table has four entries, each capable of describing one disk partition. Each entry includes

- A partition type (e.g. Primary DOS partition, UNIX partition).
- An ACTIVE indication (is this the one we boot from).
- The disk address where that partition starts and ends.
- The number of sectors contained within that partition.

Partn	Type	Active	Start (C:H:S)	End (C:H:S)	Start (logical)	Size (sectors)
-------	------	--------	---------------	-------------	-----------------	----------------

1	LINUX	True	1:0:0	199:7:49	400	79,600
2	Windows NT		200:0:0	349:7:49	80,000	60,000
3	FAT 32		350:0:0	399:7:49	140,000	20,000
4	NONE					

In older versions of DOS the starting/ending addresses were specified as cylinder/sector/head. As disks got larger, this became less practical, and they moved to logical block numbers.

The addition of disk partitioning also changed the structure of the boot record. The first sector of a disk contains the Master Boot Record (MBR) which includes the FDISK table, and a bootstrap that finds the active partition, and reads in its first sector (Partition Boot Record). Most people (essentially everyone but Bill Gates :-) make their MBR bootstrap ask what system you want to boot from, and boot the active one by default after a few seconds. This gives you the opportunity to choose which OS you want to boot. Microsoft makes this decision for you ... you want to boot Windows.

The structure of the Partition Boot Record is entirely operating system and file system specific ... but for DOS FAT file system partitions, it includes a BIOS Parameter block as described above.

#### 4. File Descriptors (directories)

In keeping with their desire for simplicity, DOS file systems combine both file description and file naming into a single file descriptor (directory entries). A DOS directory is a file (of a special type) that contains a series of fixed sized (32 byte) directory entries. Each entry describes a single file:

- an 11-byte name (8 characters of base name, plus a 3 character extension).
- a byte of attribute bits for the file, which include:
  - Is this a file, or a sub-directory.
  - Has this file changed since the last backup.
  - Is this file hidden.
  - Is this file read-only.
  - Is this a system file.
  - Does this entry describe a volume label.
- times and dates of creation and last modification, and date of last access.
- a pointer to the first logical block of the file. (This field is only 16 bits wide, and so when Microsoft introduced the FAT32 file system, they had to put the high order bits in a different part of the directory entry).
- the length (number of valid data bytes) in the file.

Name (8+3)	Attributes	Last Changed	First Cluster	Length
.	DIR	08/01/03 11:15:00	61	2,048
..	DIR	06/20/03 08:10:24	1	4,096
MARK	DIR	10/15/04 21:40:12	130	1,800
README.TXT	FILE	11/02/04 04:27:36	410	31,280

If the first character of a files name is a NULL (0x00) the directory entry is unused. The special character (0xE5) in the first character of a file name is used to indicate that a directory entry describes a deleted file. (See the section on Garbage collection below)

##### Note on times and dates:

DOS stores file modification times and dates as a pair of 16-bit numbers:

- 7 bits of year, 4 bits of month, 5 bits of day of month
- 5 bits of hour, 6 bits of minute, 5 bits of seconds (x2).

All file systems use dates relative to some epoch (time zero). For DOS, the epoch is midnight, New Year's Eve, January 1, 1980. A seven bit field for years means that the DOS calendar only runs til 2107. Hopefully, nobody will still be using DOS file systems by then :-)

## 5. Links and Free Space (File Allocation Table)

Many file systems have very compact (e.g. bitmap) free lists, but most of them use some per-file data structure to keep track of which blocks are allocated to which file. The DOS File Allocation Table is a relatively unique design. It contains one entry for each logical block in the volume. If a block is free, this is indicated by the FAT entry. If a block is allocated to a file, the FAT entry gives the logical block number of the **next** logical block in the file.

### 5.1 Cluster Size and Performance

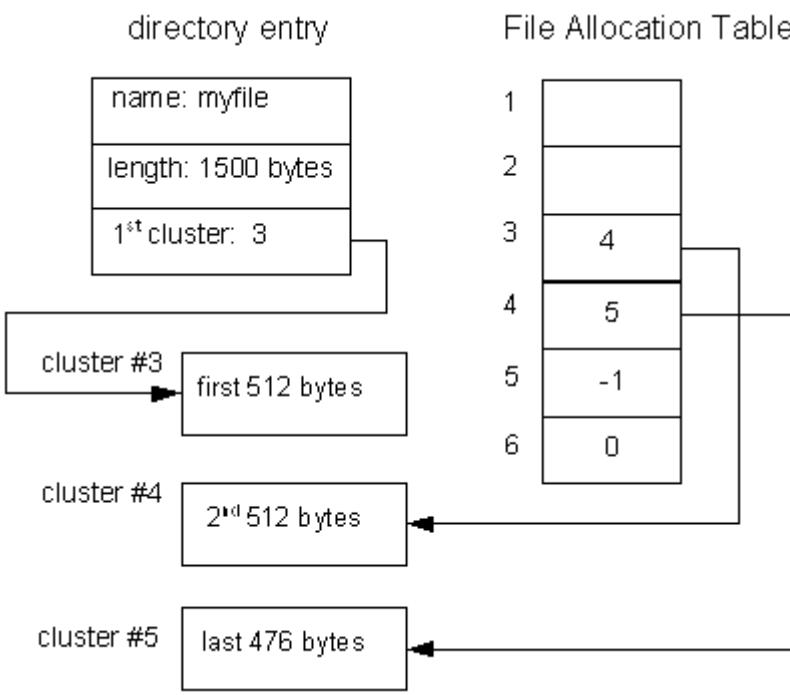
Space is allocated to files, not in (physical) blocks, but in (logical) multi-block clusters. The number of clusters per block is determined when the file system is created.

Allocating space to files in larger chunks improves I/O performance, by reducing the number of operations required to read or write a file. This comes at the cost of higher internal fragmentation (since, on average, half of the last cluster of each file is left unused). As disks have grown larger, people have become less concerned about internal fragmentation losses, and cluster sizes have increased.

The maximum number of clusters a volume can support depends on the width of the FAT entries. In the earliest FAT file systems (designed for use on floppies, and small hard drives). An 8-bit wide FAT entry would have been too small ( $256 * 512 = 128K$  bytes) to describe even the smallest floppy, but a 16-bit wide FAT entry would have been ludicrously large (8-16 Megabytes) ... so Microsoft compromised and adopted 12-bit wide FAT entries (two entries in three bytes). These were called FAT-12 file systems. As disks got larger, they created 2-byte wide (FAT-16) and 4-byte wide (FAT-32) file systems.

### 5.2 Next Block Pointers

A file's directory entry contains a pointer to the first cluster of that file. The File Allocation Table entry for that cluster tells us the cluster number **next** cluster in the file. When we finally get to the last cluster of the file, its FAT entry will contain a -1, indicating that there is no next block in the file.



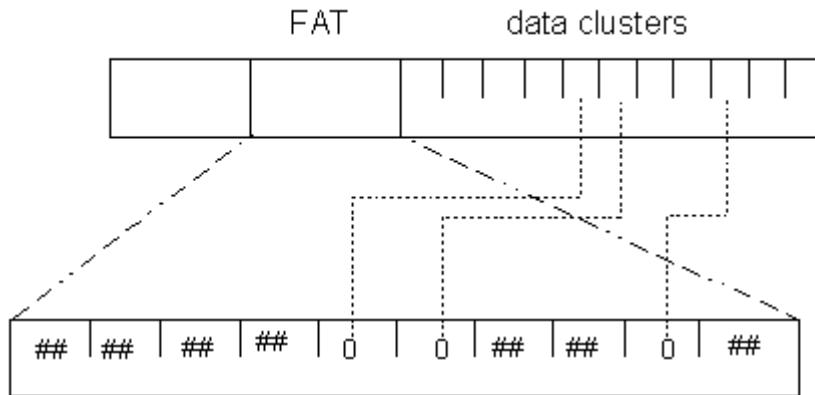
The "next block" organization of the FAT means that in order to figure out what physical cluster is the third logical block of a file, must know the physical cluster number of the second logical block. This is not usually a problem, because almost all file access is sequential (reading the first block, and then the second, and then the third ...).

If we had to go to disk to re-read the FAT each time we needed to figure out the next block number, the file system would perform very poorly. Fortunately, the FAT is so small (e.g. 512 bytes per megabyte of file system) that the entire FAT can be kept in memory as long as a file system is in use. This means that successor block numbers can be looked up without the need to do any additional disk I/O. It is easy to imagine

### 5.3 Free Space

The notion of "next block" is only meaningful for clusters that are allocated to a file ... which leaves us free to use the FAT entries associated with free clusters as a free indication. Just as we reserved a value (-1) to mean **end of file** we can reserve another value (0) to mean **this cluster is free**.

To find a free cluster, one has but to search the FAT for an entry with the value -2. If we want to find a free cluster near the clusters that are already allocated to the file, we can start our search with the FAT entry after the entry for the first cluster in the file.



Each FAT entry corresponds to one data cluster

A FAT entry of 0 means the corresponding data cluster is not allocated to any file.

## 5.4 Garbage Collection

Older versions of FAT file systems did not bother to free blocks when a file was deleted. Rather, they merely crossed out the first byte of the file name in the directory entry (with the reserved value 0xE5). This had the advantage of greatly reducing the amount of I/O associated with file deletion ... but it meant that DOS file systems regularly ran out of space.

When this happened, they would initiate garbage collection. Starting from the root directory, they would find every "valid" entry. They would follow the chain of next block pointers to determine which clusters were associated with each file, and recursively enumerate the contents of all sub-directories. After completing the enumeration of all allocated clusters, they inferred that any cluster not found in some file was free, and marked them as such in the File Allocation Table.

This "feature" was probably motivated by a combination of laziness and a desire for performance. It did, however, have an advantage. Since clusters were not freed when files were deleted, they could not be reallocated until after garbage collection was performed. This meant that it might be possible to recover the contents of deleted files for quite a while. The opportunity this created was large enough to enable Peter Norton to start a very successful company.

## 6. Descendents of the DOS file system

The DOS file system has evolved with time. Not only have wider (16- and 32-bit) FAT entries been used to support larger disks, but other features have been added. The last stand-alone DOS product was DOS 6.x. After this, all DOS support was under Windows, and along with the change to Windows came an enhanced version of the FAT file system called Virtual FAT (or simply VFAT).

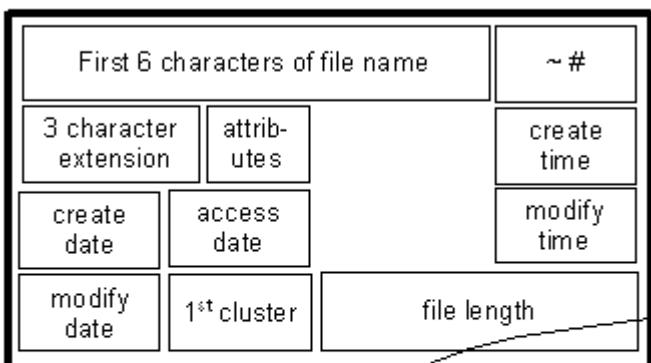
### 6.1 Long File Names

Most DOS and Windows systems were used for personal productivity, and their users didn't demand much in the way of file system features. Their biggest complaints were about the 8+3 file names. Windows users demanded longer and mixed-case file names.

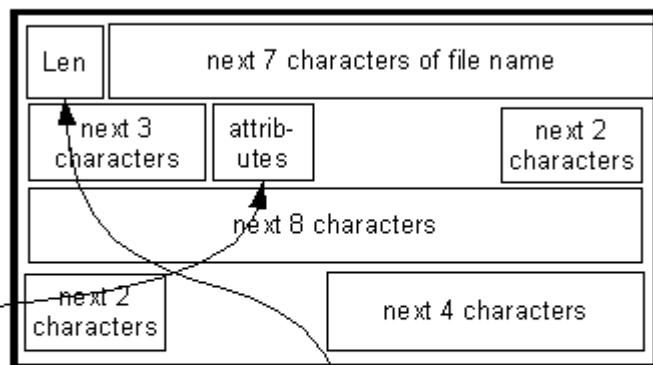
The 32 byte directory entries didn't have enough room to hold longer names, and changing the format of DOS directories would break hundreds or even thousands of applications. It wasn't merely a matter of importing files from old systems to new systems. DOS diskettes are commonly used to carry files between various systems ... which means that old systems still had to be able to read the new directories. They had to find a way to support longer file names without making the new files unreadable by older systems.

The solution they came up was to put the extended filenames in additional (auxiliary) directory entries. Each file would be described by an old-format directory entry, but supplementary directory entries (following the primary directory entry) could provide extensions to the file name. To keep older systems from being confused by the new directory entries, they were tagged with a very unusual set of attributes (hidden, system, read-only, volume label). Older systems would ignore new entries, but would still be able to access new files under their 8+3 names in the old-style directory entries. New systems would recognize the new directory entries, and be able to access files by either the long or the short names.

Primary (old style) directory entry



Secondary (continuation) directory entry



Attributes (Read Only, Hidden, System, Volume) identify this as an continuation directory entry. Older systems will ignore such entries.

Length field says how many more bytes of name are contained in this entry.

The addition of long file names did create one problem for the old directory entries. What would you do if you had two file names that differed only after the 8th character (e.g. datafileread.c and datafilewrite.c)? If we just used the first 8 characters of the file name in the old directory entries (datafile), we would have two files with the same name, and this is illegal. For this reason, the the short file names are not merely the first eight characters of the long file names. Rather, the last two bytes of the short name were merely made unambiguous (e.g. "~1", "~2", etc).

## 6.2 Alternate/back-up FATs

The File Allocation Table is a very concise way of keeping track of all of the next-block pointers in the file system. If anything ever happened to the File Allocation Table, the results would be disastrous. The directory entries would tell us where the first blocks of all files were, but we would have no way of figuring out where the remainder of the data was.

Events that corrupt the File Allocation Table are extremely rare, but the consequences of such an incident are catastrophic. To protect users from such eventualities, MicroSoft added support for alternate FATs. Periodically, the primary FAT would be copied to one of the pre-reserved alternat FAT locations. Then if something bad happened to the primary FAT, it would still be possible to read most files (files created before the copy) by using the back-up FAT. This is an imperfect solution, as losing new files is bad ... but losing all files is worse.

## 6.3 ISO 9660

When CDs were proposed for digital storage, everyone recognized the importance of a single standard file system format. Dueling formats would raise the cost of producing new products ... and this would be a lose for everyone. To respond to this need, the International Standards Organization chartered a sub-committee to propose such a standard.

The failings of the DOS file system were widely known by this time, but, as the most widely used file system on the planet, the committee members could not ignore it. Upon examination, it became clear that the most ideomatic features of the DOS file system (the File Allocation Table) were irrelevant to a CDROM file system (which is written only once):

- We don't need to keep track of the free space on a CD ROM. We write each file contiguously, and the next file goes immediately after the last one.
- Because files can be written contiguously, we don't need any "next block" pointers. All we need to know about a file is where its first block resides.

It was decided that ISO 9660 file systems would (like DOS file systems) have tree structured directory hierarchies, and that (like DOS) each directory entry would describe a single file (rather than having some auxiliary data structure like and I-node to do this). 9660 directory entries, like DOS directory entries, contain:

- file name (within the current directory)
- file type (e.g. file or directory)
- location of the file's first block
- number of bytes contained in the file
- time and date of creation

They did, however, learn from DOS's mistakes:

- Realizing that new information would be added to directory entries over time, they made them variable length. Each directory entry begins with a length field (giving the number of bytes in this directory entry, and thus the number of bytes until the next directory entry).
- Recognizing the need to support long file names, they also made the file name field in each entry a variable length field.
- Recognizing that, over time, people would want to associate a wide range of attributes with files, they also created a variable length extended attributes section after the file name. Much of this section has been left unused, but they defined several new attributes for files:
  - file owner
  - owning group
  - permissions
  - creation, modification, effective, and expiration times
  - record format, attributes, and length information

But, even though 9660 directory entries include much more information than DOS directory entries, it remains that 9660 volumes resemble DOS file systems much more than they resemble any other file system format. And so, the humble DOS file system is reborn in a new generation of products.

## 7. Summary

DOS file systems are very simple, They don't support multiple links to a file, or symbolic links, or even multi-user access control. They are also and very economical in terms of the space they take up. Free block lists, and file block pointers are combined into a single (quite compact) File Allocation Table. File descriptors are incorporated into the directory entries. And for all of these limitations, they are probably the most widely used

file system format on the planet. Despite their primitiveness, DOS file systems were used as the basis for much newer CD ROM file system designs.

What can we infer from this? That most users don't need a lot of fancy features, and that the DOS file system (primitive as it may be) covers their needs pretty well.

It is also noteworthy that when Microsoft was finally forced to change the file system format to get past the 8.3 upper case file name limitations, they chose to do so with a klugy (but upwards compatible) solution using additional directory entries per file. The fact that they chose such an implementation clearly illustrates the importance of maintaining media interchangability with older systems. This too is a problem that all (successful) file systems will eventually face.

## 8. References

DOS file system information

- PC Guide's [Overview of DOS FAT file systems](#). (this is a pointer to the long filename article ... but the entire library is nothing short of excellent).
- Free BSD sources, PCFS implementation, [BIOS Parameter block format](#), and (Open Solaris) [FDISK table format](#).
- Free BSD sources, PCFS implementation, [Directory Entry format](#)

9660 file system information

- Wikipedia Introduction to [ISO 9660 file systems](#)
- Free BSD sources, ISOFS implementation, [ISO 9660 data structures](#).

## Cluster

There are many different types of clusters. Common types include:

- load sharing clusters, which divide work among the members.
- high availability clusters, where back-up nodes take over when primary nodes fail.
- information sharing clusters, which ensure the dissemination of information throughout a network.

There are so many different goals and architectures that Greg Phister (in "In Search of Clusters") came to the conclusion that it is very difficult to even define the term. About the only thing we can say for certain is that a cluster is a networked connection of nodes, all of whom agree that they are part of a cluster.

## Membership

If a cluster is defined as a networked connection of nodes who consider themselves to be participants in the cluster, then obviously "membership" is a key concept.

We can distinguish two types of membership:

- potential, eligible or designated members
- active or currently participating members

This distinction is important because only active members can communicate with one-another. Thus it is that the term "membership" is most commonly used to describe only the currently participating members.

It is very important to know who the current (active) cluster members are. We may, for instance, be required to make sure that each of them has been informed of some operation before we are allowed to perform it. Cluster membership often comes with responsibilities (e.g. a guarantee to respond to certain requests within a certain period if time). Thus it is vital that we know when nodes enter and leave the cluster.

In most clusters, a node has to be explicitly configured or provisioned into the cluster ... so that the set of potential members is well known, and perhaps even closed to new members. There are, however, some types of clusters where any node is welcome to join at any time.

## Node Redundancy

In a clustered system, work is divided among the active members. To reduce distributed synchronization, it is common to partition the work (e.g. designate each server responsible for a certain subset (e.g. a file system, a range of keys, etc) of requests, and route all requests to their designated owner). In such systems, we can talk about *primaries* (the designated owners) and *secondaries* (nodes who are prepared to take over for a primary if he fails).

There are three fundamentally different approaches to take to high availability:

- Active/Stand-By

The system is divided into *active* and *stand-by* nodes. The incoming requests are partitioned among the active nodes. The stand-by nodes are idle until an active node fails, at which point a stand-by node takes over his work.

- Active/Active

The incoming requests are partitioned among all of the available nodes. If one node fails, his work will be redistributed among the survivors.

An active/active architecture achieves better resource utilization, and so may be more cost-effective. But when a failure occurs, the load on the surviving nodes is increased and so performance may suffer. An active/stand-by architecture normally has idle capacity, but may not suffer any performance degradation after a failure.

We can also look at how quickly a successor is able to take over a failed node's responsibilities. In some architectures, all operations are mirrored to the secondaries, enabling them to very quickly assume the primary role. Such secondaries are called *hot standbys*. In other systems, the secondary waits to be notified of the primary's failure, after which it opens and reads the failed primary's last check-point or journal. The former approach results in more network and CPU load, but much faster fail-overs. The latter approach consumes fewer resources during normal operation, but may take much longer to resume service after a primary has failed.

## Heart Beat

Ideally nodes will announce the fact that they are joining the cluster, or are about to leave it. This is not always the case:

1. a system may crash.
2. the clustering applications may crash.
3. a node may become so busy that the clustering applications cannot run.
4. a network interface or link may fail.

Since we cannot be sure that member will notify the other members before he leaves the cluster, we need a means of detecting a member who has dropped unexpectedly out of the cluster. The most common technique is called a "heart beat". A heart beat is a message, regularly exchanged between cluster members. These may be special messages, or they may be piggy-backed on other traffic. If too much time passes without our having received a heart-beat from some node, we will assume that node has failed, and is no longer an active cluster member.

The failure of a node may (in some clusters) have serious consequences (e.g. the freeing of all resources allocated to that node, and the aborting of all in progress transactions from that node). To prevent "false alarms", many systems perform heart-beats over multiple channels, or have a back-up link with which they attempt to confirm a failure before reporting a node to be dead.

## Cluster Master and Election

It is often convenient to elect or designate one node to be the cluster master:

- Coming to a mutual agreement between multiple nodes can be a complex process (e.g. Three Phase Commits). If one node is designated a cluster master, that node can serve as a central point of synchronization and/or control for operations in the cluster.
- Rather than requiring all nodes to heart-beat one-another, it is more economical to simply have all nodes heart-beat with the cluster master. He will detect the failure of any other node, and all nodes will detect his failure.

The election of a cluster master may, itself, be a complex process ... but having performed that process may eliminate the need for any further negotiations. There are numerous well established election/concensus algorithms. One of the best known is Leslie Lamport's [Paxos algorithm](#).

## Split Brain

A pathological network failure might divide a cluster into multiple sub-clusters, which cannot communicate with one-another. Such an event is sometimes referred to as a "partitioning" of the network. If the cluster manages critical resources (e.g. a database or nuclear warhead), it is possible that the independent sub-clusters will all continue operating and make independent but incompatible decisions. Such a condition is called "split-brain" (as if two halves of our brain were working independently and at cross-purposes).

There are two standard approaches to preventing "split-brain":

- a. quorum
- b. voting devices

## Quorum

If there are  $N$  potential members in a cluster, we can build in a rule that says a cluster cannot form with fewer than  $(N/2)+1$  members. This accomplishes two purposes:

- It makes it impossible for any partitioning to result in two viable sub-clusters (because  $N$  nodes cannot be divided into two groups that both contain at least  $(N/2)+1$  nodes).
- It ensures that any decision made (and persisted) by this quorum will be remembered by any future quorum (because any group of  $(N/2)+1$  nodes will have at least one member in common with every other group of  $(N/2)+1$  nodes that has ever existed).

The problem with using a numerical quorum is that if  $(N/2)+1$  nodes have been damaged, it will be impossible for the surviving nodes to form a new cluster ... even if there is no split-brain.

## Voting Devices

If there is a single piece of hardware in the cluster, that must be present for the cluster to function, and that can only be owned by one node, that device can be used as a voting device.

Consider, for instance, a shared disk. If that disk is absolutely required to provide service, a node that no longer has access to that disk cannot provide service (and hence is not eligible to form a cluster). But what if two nodes can both talk to the disk, but cannot communicate with one-another? They may be able to use the disk as a voting device ... e.g. by writing a recent time-stamp into a well-known block.

Some clusters include resources that can easily serve as voting devices. There are also specially built (very reliable) voting devices that exist solely for this purpose. If there is a voting device, a cluster could be formed by a single node ... because the voting device would prevent split-brain.

## Fencing

What if, you were not only suffering from schizophrenia, but the other side of your brain had actually gone rogue, and was trying to commit acts of mayhem against you and others? In some clustered systems, it must be assumed that if a node has fallen out of the cluster, he is no longer trust-worthy ... and must be "fenced-out" of the cluster. There are two common approaches to fencing:

- reservable devices  
Some devices can be told which interface to listen to, and not to listen to the other interface. This is often done with dual-ported disks. The node that has seized the quorum device will then instruct the quorum device not to accept commands from any other node.
- remote power control  
Some clustered systems come with remote power controllers, and a node that has seized control of the

cluster from a previous (apparently failed) cluster master will often power-off or reset the previous master, to ensure that he does not continue to vie for control of the cluster and its resources.

# Health Monitoring and Recovery

## Introduction

Suppose that a system seems to be wedged ... it has not recently made any progress. How would we determine if the system was deadlocked?

- identify all of the blocked processes.
- identify the resource on which each process is blocked.
- identify the owner of each blocking resource.
- determine whether or not the implied graph contains any loops.

How would we determine that the system might be wedged, so that we could invoke deadlock analysis? It may not be possible to identify the owners of all of the involved resources, or even all of the resources. Worse still, a process may not actually be blocked, but merely waiting for a message or event (that has, for some reason, not yet been sent). And if we did determine that a deadlock existed, what would we do? Kill a random process? This might break the circular dependency, but would the system continue to function properly after such an action? Formal deadlock detection in real systems is:

Formal deadlock detection in real systems ...

- a. is difficult to perform
- b. is inadequate to diagnose most hangs
- c. does not tell us how to fix the problem

Fortunately there is a simpler technique, that is far more effective at detecting, diagnosing, and repairing a much wider range of problems: health monitoring and managed recovery.

## Health Monitoring

We said that we could invoke deadlock detection whenever we thought that the system might not be making progress. How could we know whether or not the system was making progress? There are many ways to do this:

- by having an internal monitoring agent watch message traffic or a transaction log to determine whether or not work is continuing
- by having the service send periodic heart-beat messages to a health monitoring service.
- by having an external health monitoring service send periodic test requests to the service that is being monitored, and ascertain that they are being responded to correctly and in a timely fashion.

Any of these techniques could alert us of a potential deadlock, livelock, loop, or a wide range of other failures. But each of these techniques has different strengths and weaknesses:

- heart beat messages can only tell us that the node and application are still up and running. They cannot tell us if the application is actually serving requests.
- an external health monitoring service can determine whether or not the monitored application is responding to requests. But this does not mean that some other requests have not been deadlocked or otherwise wedged.
- an internal monitoring agent might be able to monitor logs or statistics to determine that the service is processing requests at a reasonable rate (and perhaps even that no requests have been waiting too long). But if the internal monitoring agent fails, it may not be able to detect and report errors.

Many systems use a combination of these methods:

- the first line of defense is an internal monitoring agent that closely watches key applications to detect failures and hangs.
- if the internal monitoring agent is responsible for sending heart-beats (or health status reports) to a central monitoring agent, a failure of the internal monitoring agent will be noticed by the central monitoring agent.
- an external test service that periodically generates test transactions provides an independent assessment that might include external factors (e.g. switches, load balancers, network connectivity) that would not be tested by the internal and central monitoring services.

## Managed Recovery

Suppose that some or all of these monitoring mechanisms determine that a service has hung or failed. What can we do about it? Highly available services must be designed for restart, recovery, and fail-over:

- The software should be designed so that any process in the system can be killed and restarted at any time. When a process restarts, it should be able to reestablish communication with the other processes and resume working with minimal disruption.
- The software should be designed to support multiple levels of restart. Examples might be:
  - warm-start ... restore the last saved state (from a database or from information obtained from other processes) and resume service where we left off.
  - cold-start ... ignore any saved state (which may be corrupted) and restart new operations from scratch.
- The software might also be designed for progressively escalating scope of restarts:
  - restart only a single process, and expect it to resync with the other processes when it comes back up.
  - restart all of the software on a single node.
  - restart a group of nodes, or the entire system.

Designing software in this way gives us the opportunity to begin with minimal disruption, restarting only the process that seems to have failed. In most cases this will solve the problem, but perhaps:

- process A failed as a result of an incorrect request received from process B.
- the operation that caused process A to fail is still listed in the database, and when process A restarts, it may attempt to re-try the same operation and fail again.
- the operation that caused process A to fail may have been mirrored to other systems, that will also experience or cause additional failures.

For all of these reasons it is desirable to have the ability to escalate to progressively more complete restarts of a progressively wider range of components.

## False Reports

Ideally a problem will be found by the internal monitoring agent on the affected node, which will automatically trigger a restart of the affected software on that node. Such prompt local action has the potential to fix the problem before other nodes even notice that there was a problem.

But suppose a central monitoring service notes that it has not received a heart-beat from process A. What might this mean?

- It might mean that the node has failed.
- It might mean that the process has failed.
- It might mean that the system is loaded and the heart-beat message was delayed.
- It might mean that a network error prevented or delayed the delivery of a heart-beat message.

Declaring a process to have failed can potentially be a very expensive operation. It might cause the cancellation and retransmission of all requests that had been sent to the failed process or node. It might cause other servers to start trying to recover work-in-progress from the failed process or node. And this recovery might involve a great deal of network traffic and system activity. We don't want to start an expensive fire-drill unless we are pretty certain that a process has failed.

- the best option would be for a failing process to detect its own problem, inform its partners, and shut-down cleanly.
- if the failure is detected by a missing heart-beat, it may be wise to wait until multiple heart-beat messages have been missed before declaring the process to have failed.
- in some cases, we might want to wait for multiple other processes/nodes to complain.

But there is a trade-off here. If we do not take the time to confirm suspected failures, we may suffer unnecessary service disruptions from forcing fail-overs from healthy servers. On the other hand, if we wait too long before initiating fail-overs, we are prolonging the service outage. These so-called "mark-out thresholds" often require a great deal of tuning.

## Other Managed Restarts

As we consider failure and restart, there are two other interesting cases to note:

- non-disruptive rolling upgrades ... if a system is capable of operating without some of its nodes, it is possible to achieve non-disruptive rolling software upgrades. We take nodes down, one-at-a-time, upgrade each to a new software release, and then reintegrate them into the service. There are two tricks associated with this:
  - the new software must be up-wards compatible with the old software, so that new nodes can interoperate with old ones.
  - if the rolling upgrade does not seem to be working, there needs to be an automatic *fall-back* option to return to the previous (working) release.
  -
- prophylactic reboots ... It has long been observed that many software systems become slower and more error prone the longer they run. The most common problem is memory leaks, but there are other types of bugs that can cause software systems to degrade over time. The right solution is probably to find and fix the bugs ... but many organizations seem unable to do this. One popular alternative is to automatically restart every system at a regular interval (e.g. a few hours or days).

If a system can continue operating in the face of node failures, it should be fairly simple to shut-down and restart nodes one at a time, on a regular schedule.

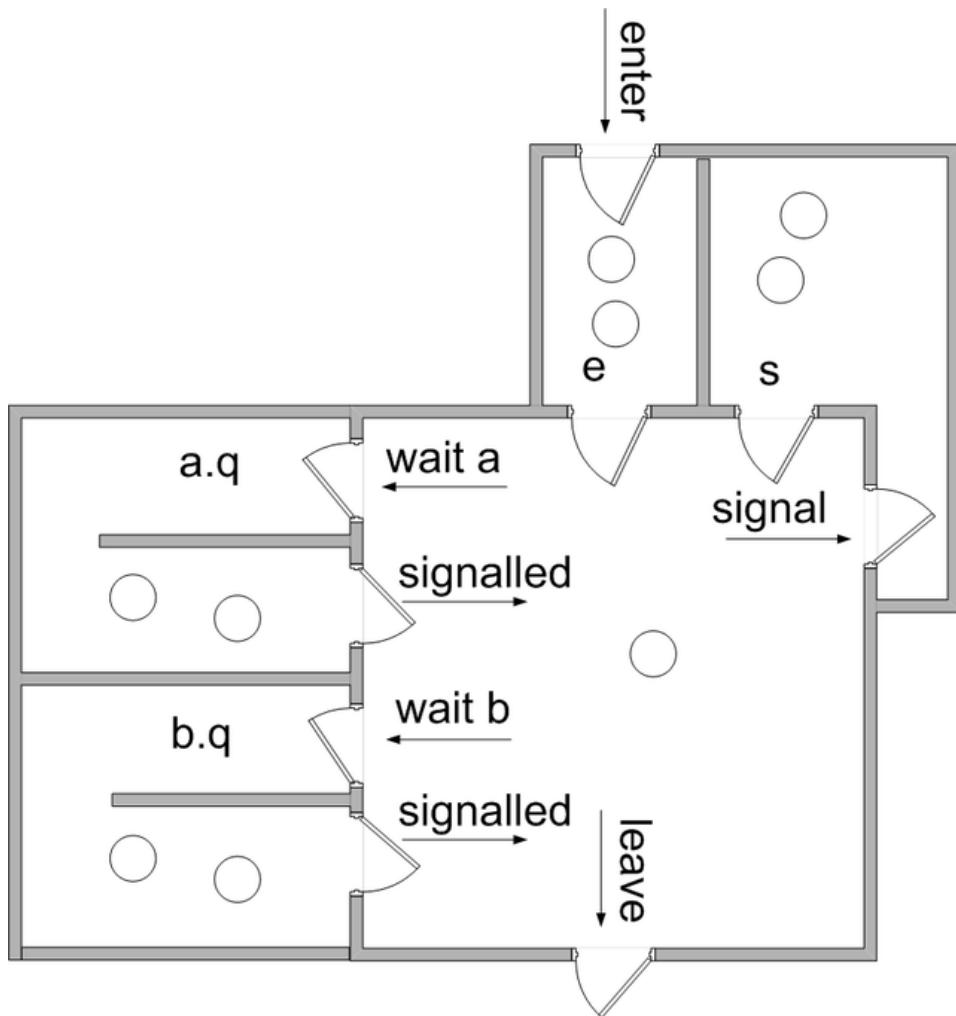
## Monitor (synchronization)



Connected to:

[Per Brinch Hansen Thread \(computer science\)](#) [Thread \(computing\)](#)

From Wikipedia, the free encyclopedia



This article contains instructions, advice, or how-to content. The purpose of Wikipedia is to present facts, not to train. Please help improve this article either by rewriting the how-to content or by moving it to Wikiversity, Wikibooks or Wikivoyage. (January 2014)

In [concurrent programming](#), a **monitor** is a synchronization construct that allows [threads](#) to have both [mutual exclusion](#) and the ability to wait (block) for a certain condition to become true. Monitors also have a mechanism for signalling other threads that their condition has been met. A monitor consists of a [mutex \(lock\)](#) object and **condition variables**. A **condition variable** is basically a container of threads that are waiting for a certain condition. Monitors provide a mechanism for threads to temporarily give up exclusive access in order to wait for some condition to be met, before regaining exclusive access and resuming their task.

Another definition of **monitor** is a **thread-safe class, object, or module** that uses wrapped [mutual exclusion](#) in order to safely allow access to a method or variable by more than one [thread](#). The defining characteristic of a monitor is that its methods are executed with [mutual exclusion](#): At each point in

time, at most one thread may be executing any of its [methods](#). Using a condition variable(s), it can also provide the ability for threads to wait on a certain condition (thus using the above definition of a "monitor"). For the rest of this article, this sense of "monitor" will be referred to as a "thread-safe object/class/module".

Monitors were invented by [Per Brinch Hansen](#)<sup>[1]</sup> and [C. A. R. Hoare](#),<sup>[2]</sup> and were first implemented in [Brinch Hansen's Concurrent Pascal](#) language.<sup>[3]</sup>

## Mutual exclusion

As a simple example, consider a thread-safe object for performing transactions on a bank account:

```
monitor class Account {
    private int balance := 0
    invariant balance >= 0

    public method boolean withdraw(int amount)
        precondition amount >= 0
    {
        if balance < amount {
            return false
        } else {
            balance := balance - amount
            return true
        }
    }

    public method deposit(int amount)
        precondition amount >= 0
    {
        balance := balance + amount
    }
}
```

While a thread is executing a method of a thread-safe object, it is said to *occupy* the object, by holding its [mutex \(lock\)](#). Thread-safe objects are implemented to enforce that *at each point in time, at most one thread may occupy the object*. The lock, which is initially unlocked, is locked at the start of each public method, and is unlocked at each return from each public method.

Upon calling one of the methods, a thread must wait until no other thread is executing any of the thread-safe object's methods before starting execution of its method. Note that without this mutual exclusion, in the present example, two threads could cause money to be lost or gained for no reason. For example, two threads withdrawing 1000 from the account could both return true, while causing the balance to drop by only 1000, as follows: first, both threads fetch the current balance, find it greater than 1000, and subtract 1000 from it; then, both threads store the balance and return.

The [syntactic sugar](#) "monitor class" in the above example is implementing the following basic representation of the code, by wrapping each function's execution in mutexes:

```
class Account {
    private lock myLock

    private int balance := 0
    invariant balance >= 0

    public method boolean withdraw(int amount)
        precondition amount >= 0
    {
        myLock.acquire()
        try {
            if balance < amount {
                return false
            } else {
                balance := balance - amount
                return true
            }
        } finally {
            myLock.release()
        }
    }

    public method deposit(int amount)
        precondition amount >= 0
    {
        myLock.acquire()
        try {
            balance := balance + amount
        } finally {
            myLock.release()
        }
    }
}
```

## Condition variables

### Problem statement

For many applications, mutual exclusion is not enough. Threads attempting an operation may need to wait until some condition  $P$  holds true. A [busy waiting](#) loop

```
while not( P ) do skip
```

will not work, as mutual exclusion will prevent any other thread from entering the monitor to make the condition true. Other "solutions" exist such as having a loop that unlocks the monitor, waits a certain amount of time, locks the monitor and check for the condition  $P$ . Theoretically, it works and will not deadlock, but issues arise. It's hard to decide an appropriate amount of waiting time, too small and the thread will hog the CPU, too big and it will be apparently unresponsive. What is needed is a way to signal the thread when the condition  $P$  is true (or *could* be true).

## Case study: classic bounded producer/consumer problem

A classic concurrency problem is that of the **bounded producer/consumer**, in which there is a [queue](#) or [ring buffer](#) of tasks with a maximum size, with one or more threads being "producer" threads that add tasks to the queue, and one or more other threads being "consumer" threads that take tasks out of the queue. The queue is assumed to be non-thread-safe itself, and it can be empty, full, or between empty and full. Whenever the queue is full of tasks, then we need the producer threads to block until there is room from consumer threads dequeuing tasks. On the other hand, whenever the queue is empty, then we need the consumer threads to block until more tasks are available due to producer threads adding them.

As the queue is a concurrent object shared between threads, accesses to it must be made [atomic](#), because the queue can be put into an **inconsistent state** during the course of the queue access that should never be exposed between threads. Thus, any code that accesses the queue constitutes a [critical section](#) that must be synchronized by mutual exclusion. If code and processor instructions in critical sections of code that access the queue could be **interleaved** by arbitrary [context switches](#) between threads on the same processor or by simultaneously-running threads on multiple processors, then there is a risk of exposing inconsistent state and causing [race conditions](#).

### Incorrect without synchronization

A naïve approach is to design the code with **busy-waiting** and no synchronization, making the code subject to race conditions:

```
global RingBuffer queue; // A thread-unsafe ring-buffer of tasks.

// Method representing each producer thread's behavior:
public method producer(){
    while(true){
        task myTask=...; // Producer makes some new task to be added.
        while(queue.isFull()){} // Busy-wait until the queue is non-full.
        queue.enqueue(myTask); // Add the task to the queue.
    }
}

// Method representing each consumer thread's behavior:
public method consumer(){
    while(true){
        while (queue.isEmpty()){} // Busy-wait until the queue is non-empty.
        myTask=queue.dequeue(); // Take a task off of the queue.
        doStuff(myTask); // Go off and do something with the task.
    }
}
```

This code has a serious problem in that accesses to the queue can be interrupted and interleaved with other threads' accesses to the queue. The `queue.enqueue` and `queue.dequeue` methods likely have instructions to update the queue's member variables such as its size, beginning and ending positions, assignment and allocation of queue elements, etc. In addition, the `queue.isEmpty()` and `queue.isFull()` methods read this shared state as well. If producer/consumer threads are allowed to be interleaved during the calls to enqueue/dequeue, then inconsistent state of the queue can be exposed leading to race conditions. In addition, if one consumer makes the queue empty in-between another consumer's exiting the busy-wait and calling "dequeue", then the second consumer will attempt to dequeue from an empty queue leading to an error. Likewise, if a producer makes the queue full in-between another producer's exiting the busy-wait and calling "enqueue", then the second producer will attempt to add to a full queue leading to an error.

### Spin-waiting

One naive approach to achieve synchronization, as alluded to above, is to use "**spin-waiting**", in which a mutex is used to protect the critical sections of code and busy-waiting is still used, with the lock being acquired and released in-between each busy-wait check.

```
global RingBuffer queue; // A thread-unsafe ring-buffer of tasks.
global Lock queueLock; // A mutex for the ring-buffer of tasks.

// Method representing each producer thread's behavior:
public method producer(){
    while(true){
        task myTask=...; // Producer makes some new task to be added.

        queueLock.acquire(); // Acquire lock for initial busy-wait check.
        while(queue.isFull()){ // Busy-wait until the queue is non-full.
            queueLock.release();
            // Drop the lock temporarily to allow a chance for other threads
            // needing queueLock to run so that a consumer might take a task.
            queueLock.acquire(); // Re-acquire the lock for the next call to "queue.isFull()".
        }

        queue.enqueue(myTask); // Add the task to the queue.
        queueLock.release(); // Drop the queue lock until we need it again to add the next task.
    }
}
```

```

}

// Method representing each consumer thread's behavior:
public method consumer(){
    while(true){
        queueLock.acquire(); // Acquire lock for initial busy-wait check.
        while (queue.isEmpty()) { // Busy-wait until the queue is non-empty.
            queueLock.release();
            // Drop the lock temporarily to allow a chance for other threads
            // needing queueLock to run so that a producer might add a task.
            queueLock.acquire(); // Re-acquire the lock for the next call to "queue.isEmpty()".
        }
        myTask=queue.dequeue(); // Take a task off of the queue.
        queueLock.release(); // Drop the queue lock until we need it again to take off the next task.
        doStuff(myTask); // Go off and do something with the task.
    }
}

```

This method assures that inconsistent state does not occur, but wastes CPU resources due to the unnecessary busy-waiting. Even if the queue is empty and producer threads have nothing to add for a long time, consumer threads are always busy-waiting unnecessarily. Likewise, even if consumers are blocked for a long time on processing their current tasks and the queue is full, producers are always busy-waiting. This is a wasteful mechanism. What is needed is a way to make producer threads block until the queue is non-full, and a way to make consumer threads block until the queue is non-empty.

(N.B.: Mutexes themselves can also be **spin-locks** which involve busy-waiting in order to get the lock, but in order to solve this problem of wasted CPU resources, we assume that *queueLock* is not a spin-lock and properly uses a blocking lock queue itself.)

## Condition variables

The solution is to use **condition variables**. Conceptually a condition variable is a queue of threads, associated with a monitor, on which a thread may wait for some condition to become true. Thus each condition variable  $c$  is associated with an [assertion](#)  $P_c$ . While a thread is waiting on a condition variable, that thread is not considered to occupy the monitor, and so other threads may enter the monitor to change the monitor's state. In most types of monitors, these other threads may signal the condition variable  $c$  to indicate that assertion  $P_c$  is true in the current state.

Thus there are two main operations on condition variables:

- **wait**  $c$ ,  $m$ , where  $c$  is a condition variable and  $m$  is a [mutex \(lock\)](#) associated with the monitor. This operation is called by a thread that needs to wait until the assertion  $P_c$  is true before proceeding. While the thread is waiting, it does not occupy the monitor. The function, and fundamental contract, of the "wait" operation, is to do the following steps:

1. [Atomically](#):

- a. release the mutex  $m$ ,
- b. move this thread from the "ready queue" to  $c$ 's "wait-queue" (a.k.a. "sleep-queue") of threads, and
- c. sleep this thread. (Context is synchronously yielded to another thread.)

2. Once this thread is subsequently notified/signalled (see below) and resumed, then automatically re-acquire the mutex  $m$ .

Steps 1a and 1b can occur in either order, with 1c usually occurring after them. While the thread is sleeping and in  $c$ 's wait-queue, the next [program counter](#) to be executed is at step 2, in the middle of the "wait" function/[subroutine](#). Thus, the thread sleeps and later wakes up in the middle of the "wait" operation.

The atomicity of the operations within step 1 is important to avoid race conditions that would be caused by a preemptive thread switch in-between them. One failure mode that could occur if these were not atomic is a *missed wakeup*, in which the thread could be on  $c$ 's sleep-queue and have released the mutex, but a preemptive thread switch occurred before the thread went to sleep, and another thread called a signal/notify operation (see below) on  $c$  moving the first thread back out of  $c$ 's queue. As soon as the first thread in question is switched back to, its program counter will be at step 1c, and it will sleep and be unable to be woken up again, violating the invariant that it should have been on  $c$ 's sleep-queue when it slept. Other race conditions depend on the ordering of steps 1a and 1b, and depend on where a context switch occurs.

- **signal**  $c$ , also known as **notify**  $c$ , is called by a thread to indicate that the assertion  $P_c$  is true. Depending on the type and implementation of the monitor, this moves one or more threads from  $c$ 's sleep-queue to the "ready queue" or another queues for it to be executed. It is usually considered a best practice to perform the "signal"/"notify" operation before releasing mutex  $m$  that is associated with  $c$ , but as long as the code is properly designed for concurrency and depending on the threading implementation, it is often also acceptable to release the lock before signalling. Depending on the threading implementation, the ordering of this can have scheduling-priority ramifications. (Some authors instead advocate a preference for releasing the lock before signalling.) A threading implementation should document any special constraints on this ordering.
  - **broadcast**  $c$ , also known as **notifyall**  $c$ , is a similar operation that wakes up all threads in  $c$ 's wait-queue. This empties the wait-queue. Generally, when more than one predicate condition is associated with the same condition variable, the application will require **broadcast** instead of **signal** because a thread waiting for the wrong condition might be woken up and then immediately go back to sleep without waking up a thread waiting for the correct condition that just became true. Otherwise, if the predicate condition is one-to-one with the condition variable associated with it, then **signal** may be more efficient than **broadcast**.

As a design rule, multiple condition variables can be associated with the same mutex, but not vice versa. (This is a [one-to-many](#) correspondence.) This is because the predicate  $P_c$  is the same for all threads using the monitor and must be protected with mutual exclusion from all other threads that might cause the condition to be changed or that might read it while the thread in question causes it to be changed, but there may be different threads that want to wait for a different condition on the same variable requiring the same mutex to be used. In the producer-consumer example [described above](#), the queue must be protected by a unique mutex object,  $m$ . The "producer" threads will want to wait on a monitor using lock  $m$  and a condition variable  $c_{full}$  which blocks until the queue is non-full. The "consumer" threads will want to wait on a different monitor using the same mutex  $m$  but a different condition

variable `cempty` which blocks until the queue is non-empty. It would (usually) never make sense to have different mutexes for the same condition variable, but this classic example shows why it often certainly makes sense to have multiple condition variables using the same mutex. A mutex used by one or more condition variables (one or more monitors) may also be shared with code that does *not* use condition variables (and which simply acquires/releases it without any wait/signal operations), if those [critical sections](#) do not happen to require waiting for a certain condition on the concurrent data.

## Monitor usage

The proper basic usage of a monitor is:

```
acquire(m); // Acquire this monitor's lock.
while (!p) { // While the condition/predicate/assertion that we are waiting for is not true...
    wait(m, cv); // Wait on this monitor's lock and condition variable.
}
// ... Critical section of code goes here ...
signal(cv2); -- OR -- notifyAll(cv2); // cv2 might be the same as cv or different.
release(m); // Release this monitor's lock.
```

To be more precise, this is the same pseudocode but with more verbose comments to better explain what is going on:

```
// ... (previous code)
// About to enter the monitor.
// Acquire the advisory mutex (lock) associated with the concurrent data that is shared between threads,
// to ensure that no two threads can be preemptively interleaved or run simultaneously on different cores
// while executing in critical sections that read or write this same concurrent data.
// If another thread is holding this mutex, then this thread will be slept (blocked) and placed on
// m's sleep queue. (Mutex "m" shall not be a spin-lock.)
acquire(m);
// Now, we are holding the lock and can check the condition for the first time.

// The first time we execute the while loop condition after the above "acquire", we are asking,
// "Does the condition/predicate/assertion we are waiting for happen to already be true?"

while ( ! p() ) // "p" is any expression (e.g. variable or function-call) that checks the condition
                // and evaluates to boolean. This itself is a critical section, so you *MUST*
                // be holding the lock when executing this "while" loop condition!

// If this is not the first time the "while" condition is being checked, then we are asking the question,
// "Now that another thread using this monitor has notified me and woken me up and I have been
// context-switched back to, did the condition/predicate/assertion we are waiting on stay true between
// the time that I was woken up and the time that I
// re-acquired the lock inside the "wait" call in the last iteration of this loop,
// or did some other thread cause the condition to become false again in the meantime
// thus making this a spurious wakeup?

{
    // If this is the first iteration of the loop, then the answer is "no" -- the condition is not ready yet.
    // Otherwise, the answer is: the latter. This was a spurious wakeup, some other thread occurred first
    // and caused the condition to become false again, and we must wait again.

    wait(m, cv);
        // Temporarily prevent any other thread on any core from doing operations on m or cv.
        // release(m) // Atomically release lock "m" so other code using this concurrent data
        //             // can operate, move this thread to cv's wait-queue so that it will be notified
        //             // sometime when the condition becomes true, and sleep this thread.
        //             // Re-enable other threads and cores to do operations on m and cv.
        //
        // Context switch occurs on this core.
        //
        // At some future time, the condition we are waiting for becomes true,
        // and another thread using this monitor (m, cv) does either a signal/notify
        // that happens to wake this thread up, or a notifyAll that wakes us up, meaning
        // that we have been taken out of cv's wait-queue.
        //
        // During this time, other threads may be switched to that caused the condition to become
        // false again, or the condition may toggle one or more times, or it may happen to
        // stay true.
        //
        // This thread is switched back to on some core.
        //
        // acquire(m) // Lock "m" is re-acquired.

    // End this loop iteration and re-check the "while" loop condition to make sure the predicate is
    // still true.

}

// The condition we are waiting for is true!
// We are still holding the lock, either from before entering the monitor or from the
// last execution of "wait".

// Critical section of code goes here, which has a precondition that our predicate
// must be true.
// This code might make cv's condition false, and/or make other condition variables'
// predicates true.
```

```

// Call signal/notify or notifyAll, depending on which condition variables' predicates
// (who share mutex m) have been made true or may have been made true, and the monitor semantic type
// being used.

for (cv_x in cvs_to_notify){
    notify(cv_x); -- OR -- notifyAll(cv_x);
}
// One or more threads have been woken up but will block as soon as they try
// to acquire m.

// Release the mutex so that notified thread(s) and others can enter
// their critical sections.
release(m);

```

### Solving the bounded producer/consumer problem

This section may be too technical for most readers to understand. Please help improve this section to make it understandable to non-experts, without removing the technical details. The talk page may contain suggestions. (January 2014) (Learn how and when to remove this template message)

Having introduced the usage of condition variables, let's use it to revisit and solve the classic bounded producer/consumer problem. The classic solution is to use two monitors, comprising two condition variables sharing one lock on the queue:

```

global volatile RingBuffer queue; // A thread-unsafe ring-buffer of tasks.
global Lock queueLock; // A mutex for the ring-buffer of tasks. (Not a spin-lock.)
global CV queueEmptyCV; // A condition variable for consumer threads waiting for the queue to become non-empty.
                        // Its associated lock is "queueLock".
global CV queueFullCV; // A condition variable for producer threads waiting for the queue to become non-full.
                        // Its associated lock is also "queueLock".

// Method representing each producer thread's behavior:
public method producer(){
    while(true){
        task myTask=...; // Producer makes some new task to be added.

        queueLock.acquire(); // Acquire lock for initial predicate check.
        while(queue.isFull()){ // Check if the queue is non-full.
            // Make the threading system atomically release queueLock,
            // enqueue this thread onto queueFullCV, and sleep this thread.
            wait(queueLock, queueFullCV);
            // Then, "wait" automatically re-acquires "queueLock" for re-checking
            // the predicate condition.
        }

        // Critical section that requires the queue to be non-full.
        // N.B.: We are holding queueLock.
        queue.enqueue(myTask); // Add the task to the queue.

        // Now the queue is guaranteed to be non-empty, so signal a consumer thread
        // or all consumer threads that might be blocked waiting for the queue to be non-empty:
        signal(queueEmptyCV); -- OR -- notifyAll(queueEmptyCV);

        // End of critical sections related to the queue.
        queueLock.release(); // Drop the queue lock until we need it again to add the next task.
    }
}

// Method representing each consumer thread's behavior:
public method consumer(){
    while(true){

        queueLock.acquire(); // Acquire lock for initial predicate check.
        while (queue.isEmpty()){ // Check if the queue is non-empty.
            // Make the threading system atomically release queueLock,
            // enqueue this thread onto queueEmptyCV, and sleep this thread.
            wait(queueLock, queueEmptyCV);
            // Then, "wait" automatically re-acquires "queueLock" for re-checking
            // the predicate condition.
        }

        // Critical section that requires the queue to be non-empty.
        // N.B.: We are holding queueLock.
        myTask=queue.dequeue(); // Take a task off of the queue.
        // Now the queue is guaranteed to be non-full, so signal a producer thread
        // or all producer threads that might be blocked waiting for the queue to be non-full:
        signal(queueFullCV); -- OR -- notifyAll(queueFullCV);

        // End of critical sections related to the queue.
        queueLock.release(); // Drop the queue lock until we need it again to take off the next task.

        doStuff(myTask); // Go off and do something with the task.
    }
}

```

This ensures concurrency between the producer and consumer threads sharing the task queue, and blocks the threads that have nothing to do rather than busy-waiting as shown in the aforementioned approach using spin-locks.

A variant of this solution could use a single condition variable for both producers and consumers, perhaps named "queueFullOrEmptyCV" or "queueSizeChangedCV". In this case, more than one condition is associated with the condition variable, such that the condition variable represents a weaker condition than the conditions being checked by individual threads. The condition variable represents threads that are waiting for the queue to be non-full *and* ones waiting for it to be non-empty. However, doing this would require using *notifyAll* in all the threads using the condition variable and cannot use a regular *signal*. This is because the regular *signal* might wake up a thread of the wrong type whose condition has not yet been met, and that thread would go back to sleep without a thread of the correct type getting signalled. For example, a producer might make the queue full and wake up another producer instead of a consumer, and the woken producer would go back to sleep. In the complementary case, a consumer might make the queue empty and wake up another consumer instead of a producer, and the consumer would go back to sleep. Using *notifyAll* ensures that some thread of the right type will proceed as expected by the problem statement.

Here is the variant using only one condition variable and *notifyAll*:

```

global volatile RingBuffer queue; // A thread-unsafe ring-buffer of tasks.
global Lock queueLock; // A mutex for the ring-buffer of tasks. (Not a spin-lock.)
global CV queueFullOrEmptyCV; // A single condition variable for when the queue is not ready for any thread
    // -- i.e., for producer threads waiting for the queue to become non-full
    // and consumer threads waiting for the queue to become non-empty.
    // Its associated lock is "queueLock".
    // Not safe to use regular "signal" because it is associated with
    // multiple predicate conditions (assertions).

// Method representing each producer thread's behavior:
public method producer(){
    while(true){
        task myTask=...; // Producer makes some new task to be added.

        queueLock.acquire(); // Acquire lock for initial predicate check.
        while(queue.isFull()){ // Check if the queue is non-full.
            // Make the threading system atomically release queueLock,
            // enqueue this thread onto the CV, and sleep this thread.
            wait(queueLock, queueFullOrEmptyCV);
            // Then, "wait" automatically re-acquires "queueLock" for re-checking
            // the predicate condition.
        }

        // Critical section that requires the queue to be non-full.
        // N.B.: We are holding queueLock.
        queue.enqueue(myTask); // Add the task to the queue.

        // Now the queue is guaranteed to be non-empty, so signal all blocked threads
        // so that a consumer thread will take a task:
        notifyAll(queueFullOrEmptyCV); // Do not use "signal" (as it might wake up another producer instead).

        // End of critical sections related to the queue.
        queueLock.release(); // Drop the queue lock until we need it again to add the next task.
    }
}

// Method representing each consumer thread's behavior:
public method consumer(){
    while(true){

        queueLock.acquire(); // Acquire lock for initial predicate check.
        while (queue.isEmpty()){ // Check if the queue is non-empty.
            // Make the threading system atomically release queueLock,
            // enqueue this thread onto the CV, and sleep this thread.
            wait(queueLock, queueFullOrEmptyCV);
            // Then, "wait" automatically re-acquires "queueLock" for re-checking
            // the predicate condition.
        }

        // Critical section that requires the queue to be non-full.
        // N.B.: We are holding queueLock.
        myTask=queue.dequeue(); // Take a task off of the queue.

        // Now the queue is guaranteed to be non-full, so signal all blocked threads
        // so that a producer thread will take a task:
        notifyAll(queueFullOrEmptyCV); // Do not use "signal" (as it might wake up another consumer instead).

        // End of critical sections related to the queue.
        queueLock.release(); // Drop the queue lock until we need it again to take off the next task.

        doStuff(myTask); // Go off and do something with the task.
    }
}

```

## Synchronization primitives

Implementing mutexes and condition variables requires some kind of synchronization primitive provided by hardware support that provides atomicity. Locks and condition variables are higher-level abstractions over these synchronization primitives. On a uniprocessor, disabling and enabling interrupts is a way to implement monitors by preventing context switches during the critical sections of the locks and condition variables, but this is not enough on a multiprocessor. On a multiprocessor, usually special atomic **read-modify-write** instructions on the memory such as **test-and-set**, **compare-and-swap**, etc. are used, depending on what the ISA provides. These usually require deferring to spin-locking for the internal lock state itself, but this locking is very brief. Depending on the implementation, the atomic read-modify-write instructions may lock the bus from other cores' accesses and/or prevent re-

ordering of instructions in the CPU. Here is an example pseudocode implementation of parts of a threading system and mutexes and Mesa-style condition variables, using **test-and-set** and a first-come, first-served policy. This glosses over most of how a threading system works, but shows the parts relevant to mutexes and condition variables:

### Sample Mesa-monitor implementation with Test-and-Set

This section may be too technical for most readers to understand. Please help improve this section to make it understandable to non-experts, without removing the technical details. The talk page may contain suggestions. (January 2014) (Learn how and when to remove this template message)

```
// Basic parts of threading system:  
// Assume "ThreadQueue" supports random access.  
public volatile ThreadQueue readyQueue; // Thread-unsafe queue of ready threads. Elements are (Thread*).  
public volatile global Thread* currentThread; // Assume this variable is per-core. (Others are shared.)  
  
// Implements a spin-lock on just the synchronized state of the threading system itself.  
// This is used with test-and-set as the synchronization primitive.  
public volatile global bool threadingSystemBusy=false;  
  
// Context-switch interrupt service routine (ISR):  
// On the current CPU core, preemptively switch to another thread.  
public method contextSwitchISR(){  
    if (testAndSet(threadingSystemBusy)){  
        return; // Can't switch context right now.  
    }  
  
    // Ensure this interrupt can't happen again which would foul up the context switch:  
    systemCall_disableInterrupts();  
  
    // Get all of the registers of the currently-running process.  
    // For Program Counter (PC), we will need the instruction location of  
    // the "resume" label below. Getting the register values is platform-dependent and may involve  
    // reading the current stack frame, JMP/CALL instructions, etc. (The details are beyond this scope.)  
    currentThread->registers = getAllRegisters(); // Store the registers in the "currentThread" object in memory.  
    currentThread->registers.PC = resume; // Set the next PC to the "resume" label below in this method.  
  
    readyQueue.enqueue(currentThread); // Put this thread back onto the ready queue for later execution.  
  
    Thread* otherThread=readyQueue.dequeue(); // Remove and get the next thread to run from the ready queue.  
  
    currentThread=otherThread; // Replace the global current-thread pointer value so it is ready for the next thread.  
  
    // Restore the registers from currentThread/otherThread, including a jump to the stored PC of the other thread  
    // (at "resume" below). Again, the details of how this is done are beyond this scope.  
    restoreRegisters(otherThread.registers);  
  
    // *** Now running "otherThread" (which is now "currentThread")! The original thread is now "sleeping". ***  
  
    resume: // This is where another contextSwitch() call needs to set PC to when switching context back here.  
  
    // Return to where otherThread left off.  
  
    threadingSystemBusy=false; // Must be an atomic assignment.  
    systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.  
  
}  
  
// Thread sleep method:  
// On current CPU core, a synchronous context switch to another thread without putting  
// the current thread on the ready queue.  
// Must be holding "threadingSystemBusy" and disabled interrupts so that this method  
// doesn't get interrupted by the thread-switching timer which would call contextSwitchISR().  
// After returning from this method, must clear "threadingSystemBusy".  
public method threadsleep(){  
    // Get all of the registers of the currently-running process.  
    // For Program Counter (PC), we will need the instruction location of  
    // the "resume" label below. Getting the register values is platform-dependent and may involve  
    // reading the current stack frame, JMP/CALL instructions, etc. (The details are beyond this scope.)  
    currentThread->registers = getAllRegisters(); // Store the registers in the "currentThread" object in memory.  
    currentThread->registers.PC = resume; // Set the next PC to the "resume" label below in this method.  
  
    // Unlike contextSwitchISR(), we will not place currentThread back into readyQueue.  
    // Instead, it has already been placed onto a mutex's or condition variable's queue.  
  
    Thread* otherThread=readyQueue.dequeue(); // Remove and get the next thread to run from the ready queue.  
  
    currentThread=otherThread; // Replace the global current-thread pointer value so it is ready for the next thread.  
  
    // Restore the registers from currentThread/otherThread, including a jump to the stored PC of the other thread  
    // (at "resume" below). Again, the details of how this is done are beyond this scope.  
    restoreRegisters(otherThread.registers);  
  
    // *** Now running "otherThread" (which is now "currentThread")! The original thread is now "sleeping". ***  
  
    resume: // This is where another contextSwitch() call needs to set PC to when switching context back here.  
  
    // Return to where otherThread left off.
```

```

}

public method wait(Mutex m, ConditionVariable c){
    // Internal spin-lock while other threads on any core are accessing this object's
    // "held" and "threadQueue", or "readyQueue".
    while (testAndSet(threadingSystemBusy)){}
    // N.B.: "threadingSystemBusy" is now true.

    // System call to disable interrupts on this core so that threadSleep() doesn't get interrupted by
    // the thread-switching timer on this core which would call contextSwitchISR().
    // Done outside threadSleep() for more efficiency so that this thread will be slepted
    // right after going on the condition-variable queue.
    systemCall_disableInterrupts();

    assert m.held; // (Specifically, this thread must be the one holding it.)

    m.release();
    c.waitingThreads.enqueue(currentThread);

    threadSleep();

    // Thread sleeps ... Thread gets woken up from a signal/broadcast.

    threadingSystemBusy=false; // Must be an atomic assignment.
    systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.

    // Mesa style:
    // Context switches may now occur here, making the client caller's predicate false.

    m.acquire();
}

public method signal(ConditionVariable c){

    // Internal spin-lock while other threads on any core are accessing this object's
    // "held" and "threadQueue", or "readyQueue".
    while (testAndSet(threadingSystemBusy)){}
    // N.B.: "threadingSystemBusy" is now true.

    // System call to disable interrupts on this core so that threadSleep() doesn't get interrupted by
    // the thread-switching timer on this core which would call contextSwitchISR().
    // Done outside threadSleep() for more efficiency so that this thread will be slepted
    // right after going on the condition-variable queue.
    systemCall_disableInterrupts();

    if (!c.waitingThreads.isEmpty()){
        wokenThread=c.waitingThreads.dequeue();
        readyQueue.enqueue(wokenThread);
    }

    threadingSystemBusy=false; // Must be an atomic assignment.
    systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.

    // Mesa style:
    // The woken thread is not given any priority.

}

public method broadcast(ConditionVariable c){

    // Internal spin-lock while other threads on any core are accessing this object's
    // "held" and "threadQueue", or "readyQueue".
    while (testAndSet(threadingSystemBusy)){}
    // N.B.: "threadingSystemBusy" is now true.

    // System call to disable interrupts on this core so that threadSleep() doesn't get interrupted by
    // the thread-switching timer on this core which would call contextSwitchISR().
    // Done outside threadSleep() for more efficiency so that this thread will be slepted
    // right after going on the condition-variable queue.
    systemCall_disableInterrupts();

    while (!c.waitingThreads.isEmpty()){
        wokenThread=c.waitingThreads.dequeue();
        readyQueue.enqueue(wokenThread);
    }

    threadingSystemBusy=false; // Must be an atomic assignment.
    systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.

    // Mesa style:
    // The woken threads are not given any priority.

}

class Mutex {
    protected volatile bool held=false;
}

```

```

private volatile ThreadQueue blockingThreads; // Thread-unsafe queue of blocked threads. Elements are (Thread*).

public method acquire(){
    // Internal spin-lock while other threads on any core are accessing this object's
    // "held" and "threadQueue", or "readyQueue".
    while (testAndSet(threadingSystemBusy)){}
    // N.B.: "threadingSystemBusy" is now true.

    // System call to disable interrupts on this core so that threadSleep() doesn't get interrupted by
    // the thread-switching timer on this core which would call contextSwitchISR().
    // Done outside threadSleep() for more efficiency so that this thread will be slepted
    // right after going on the lock queue.
    systemCall_disableInterrupts();

    assert !blockingThreads.contains(currentThread);

    if (held){
        // Put "currentThread" on this lock's queue so that it will be
        // considered "sleeping" on this lock.
        // Note that "currentThread" still needs to be handled by threadSleep().
        readyQueue.remove(currentThread);
        blockingThreads.enqueue(currentThread);
        threadSleep();

        // Now we are woken up, which must be because "held" became false.
        assert !held;
        assert !blockingThreads.contains(currentThread);
    }

    held=true;

    threadingSystemBusy=false; // Must be an atomic assignment.
    systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.

}

public method release(){
    // Internal spin-lock while other threads on any core are accessing this object's
    // "held" and "threadQueue", or "readyQueue".
    while (testAndSet(threadingSystemBusy)){}
    // N.B.: "threadingSystemBusy" is now true.

    // System call to disable interrupts on this core for efficiency.
    systemCall_disableInterrupts();

    assert held; // (Release should only be performed while the lock is held.)

    held=false;

    if (!blockingThreads.isEmpty()){
        Thread* unblockedThread=blockingThreads.dequeue();
        readyQueue.enqueue(unblockedThread);
    }

    threadingSystemBusy=false; // Must be an atomic assignment.
    systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.

}

struct ConditionVariable {

    volatile ThreadQueue waitingThreads;
}

```

## Semaphore

As an example, consider a thread-safe class that implements a [semaphore](#). There are methods to increment (V) and to decrement (P) a private integer  $s$ . However, the integer must never be decremented below 0; thus a thread that tries to decrement must wait until the integer is positive. We use a condition variable  $sIsPositive$  with an associated assertion of  $P_{sIsPositive} = (s > 0)$ .

```

monitor class Semaphore
{
    private int s := 0
    invariant s >= 0
    private Condition sIsPositive /* associated with s > 0 */

    public method P()
    {
        while s = 0:
            wait sIsPositive
            assert s > 0
            s := s - 1
    }
}

```

```

public method v()
{
    s := s + 1
    assert s > 0
    signal sIsPositive
}
}

```

Implemented showing all synchronization (removing the assumption of a thread-safe class and showing the mutex):

```

class Semaphore
{
    private volatile int s := 0
    invariant s >= 0
    private ConditionVariable sIsPositive /* associated with s > 0 */
    private Mutex myLock /* Lock on "s" */

    public method P()
    {
        myLock.acquire()
        while s = 0:
            wait(myLock, sIsPositive)
        assert s > 0
        s := s - 1
        myLock.release()
    }

    public method V()
    {
        myLock.acquire()
        s := s + 1
        assert s > 0
        signal sIsPositive
        myLock.release()
    }
}

```

### Monitor implemented using semaphores

Conversely, locks and condition variables can also be derived from semaphores, thus making monitors and semaphores reducible to one another:

The implementation given here is incorrect. If a thread calls wait() after signal() has been called it may be stuck indefinitely, since signal() increments the semaphore only enough times for threads already waiting.

```

public method wait(Mutex m, ConditionVariable c){

    assert m.held;

    c.internalMutex.acquire();

    c.numWaiters++;
    m.release(); // Can go before/after the neighboring lines.
    c.internalMutex.release();

    // Another thread could signal here, but that's OK because of how
    // semaphores count. If c.sem's number becomes 1, we'll have no
    // waiting time.
    c.sem.Proberen(); // Block on the CV.
    // Woken
    m.acquire(); // Re-acquire the mutex.
}

public method signal(ConditionVariable c){

    c.internalMutex.acquire();
    if (c.numWaiters > 0){
        c.numWaiters--;
        c.sem.Verhogen(); // (Doesn't need to be protected by c.internalMutex.)
    }
    c.internalMutex.release();
}

public method broadcast(ConditionVariable c){

    c.internalMutex.acquire();
    while (c.numWaiters > 0){
        c.numWaiters--;
        c.sem.Verhogen(); // (Doesn't need to be protected by c.internalMutex.)
    }
    c.internalMutex.release();
}

```

```

class Mutex {

    protected boolean held=false; // For assertions only, to make sure sem's number never goes > 1.
    protected Semaphore sem=Semaphore(1); // The number shall always be at most 1.
        // Not held <--> 1; held <--> 0.

    public method acquire(){

        sem.Proberen();
        assert !held;
        held=true;

    }

    public method release(){

        assert held; // Make sure we never Verhogen sem above 1. That would be bad.
        held=false;
        sem.Verhogen();

    }

}

class ConditionVariable {

    protected int numWaiters=0; // Roughly tracks the number of waiters blocked in sem.
        // (The semaphore's internal state is necessarily private.)
    protected Semaphore sem=Semaphore(0); // Provides the wait queue.
    protected Mutex internalMutex; // (Really another Semaphore. Protects "numWaiters".)

}

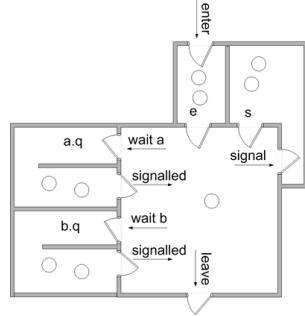
```

When a **signal** happens on a condition variable that at least one other thread is waiting on, there are at least two threads that could then occupy the monitor: the thread that signals and any one of the threads that is waiting. In order that at most one thread occupies the monitor at each time, a choice must be made. Two schools of thought exist on how best to resolve this choice. This leads to two kinds of condition variables which will be examined next:

- *Blocking condition variables* give priority to a signaled thread.
- *Nonblocking condition variables* give priority to the signaling thread.

## Blocking condition variables

The original proposals by [C. A. R. Hoare](#) and [Per Brinch Hansen](#) were for *blocking condition variables*. With a blocking condition variable, the signaling thread must wait outside the monitor (at least) until the signaled thread relinquishes occupancy of the monitor by either returning or by again waiting on a condition variable. Monitors using blocking condition variables are often called *Hoare-style* monitors or *signal-and-urgent-wait* monitors.



A Hoare style monitor with two condition variables a and b. After Buhr *et al.*

We assume there are two queues of threads associated with each monitor object

- e is the entrance queue
- s is a queue of threads that have signaled.

In addition we assume that for each condition variable c, there is a queue

- c.q, which is a queue for threads waiting on condition variable c

All queues are typically guaranteed to be [fair](#) and, in some implementations, may be guaranteed to be [first in first out](#).

The implementation of each operation is as follows. (We assume that each operation runs in mutual exclusion to the others; thus restarted threads do not begin executing until the operation is complete.)

```

enter the monitor:
    enter the method
    if the monitor is locked
        add this thread to e
        block this thread

```

```

else
    lock the monitor

leave the monitor:
    schedule
    return from the method

wait c :
    add this thread to c.q
    schedule
    block this thread

signal c :
    if there is a thread waiting on c.q
        select and remove one such thread t from c.q
        (t is called "the signaled thread")
        add this thread to s
        restart t
        (so t will occupy the monitor next)
        block this thread

schedule :
    if there is a thread on s
        select and remove one thread from s and restart it
        (this thread will occupy the monitor next)
    else if there is a thread on e
        select and remove one thread from e and restart it
        (this thread will occupy the monitor next)
    else
        unlock the monitor
        (the monitor will become unoccupied)

```

The **schedule** routine selects the next thread to occupy the monitor or, in the absence of any candidate threads, unlocks the monitor.

The resulting signaling discipline is known a "*signal and urgent wait*," as the signaler must wait, but is given priority over threads on the entrance queue. An alternative is "*signal and wait*," in which there is no **s** queue and signaler waits on the **e** queue instead.

Some implementations provide a **signal and return** operation that combines signaling with returning from a procedure.

```

signal c and return :
    if there is a thread waiting on c.q
        select and remove one such thread t from c.q
        (t is called "the signaled thread")
        restart t
        (so t will occupy the monitor next)
    else
        schedule
    return from the method

```

In either case ("signal and urgent wait" or "signal and wait"), when a condition variable is signaled and there is at least one thread on waiting on the condition variable, the signaling thread hands occupancy over to the signaled thread seamlessly, so that no other thread can gain occupancy in between. If  $P_c$  is true at the start of each **signal** c operation, it will be true at the end of each **wait** c operation. This is summarized by the following [contracts](#). In these contracts,  $I$  is the monitor's [invariant](#).

```

enter the monitor:
    postcondition I

leave the monitor:
    precondition I

wait c :
    precondition I
    modifies the state of the monitor
    postcondition  $P_c$  and  $I$ 

signal c :
    precondition  $P_c$  and  $I$ 
    modifies the state of the monitor
    postcondition I

signal c and return :
    precondition  $P_c$  and  $I$ 

```

In these contracts, it is assumed that  $I$  and  $P_c$  do not depend on the contents or lengths of any queues.

(When the condition variable can be queried as to the number of threads waiting on its queue, more sophisticated contracts can be given. For example, a useful pair of contracts, allowing occupancy to be passed without establishing the invariant, is

```

wait c :
    precondition I
    modifies the state of the monitor
    postcondition  $P_c$ 

signal c
    precondition (not empty(c) and  $P_c$ ) or (empty(c) and  $I$ )

```

```

modifies the state of the monitor
postcondition I

```

See Howard<sup>[4]</sup> and Buhr *et al.*,<sup>[5]</sup> for more).

It is important to note here that the assertion  $P_c$  is entirely up to the programmer; he or she simply needs to be consistent about what it is.

We conclude this section with an example of a thread-safe class using a blocking monitor that implements a bounded, [thread-safe stack](#).

```

monitor class SharedStack {
    private const capacity := 10
    private int[capacity] A
    private int size := 0
    invariant 0 <= size and size <= capacity
    private BlockingCondition theStackIsNotEmpty /* associated with 0 < size and size <= capacity */
    private BlockingCondition theStackIsNotFull /* associated with 0 <= size and size < capacity */

    public method push(int value)
    {
        if size = capacity then wait theStackIsNotFull
        assert 0 <= size and size < capacity
        A[size] := value ; size := size + 1
        assert 0 < size and size <= capacity
        signal theStackIsNotEmpty and return
    }

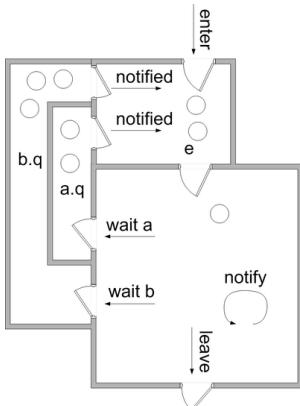
    public method int pop()
    {
        if size = 0 then wait theStackIsNotEmpty
        assert 0 < size and size <= capacity
        size := size - 1 ;
        assert 0 <= size and size < capacity
        signal theStackIsNotFull and return A[size]
    }
}

```

Note that, in this example, the thread-safe stack is internally providing a mutex, which, as in the earlier producer/consumer example, is shared by both condition variables, which are checking different conditions on the same concurrent data. The only difference is that the producer/consumer example assumed a regular non-thread-safe queue and was using a standalone mutex and condition variables, without these details of the monitor abstracted away as is the case here. In this example, when the "wait" operation is called, it must somehow be supplied with the thread-safe stack's mutex, such as if the "wait" operation is an integrated part of the "monitor class". Aside from this kind of abstracted functionality, when a "raw" monitor is used, it will *always* have to include a mutex and a condition variable, with a unique mutex for each condition variable.

## Nonblocking condition variables

With *nonblocking condition variables* (also called "*Mesa style*" condition variables or "*signal and continue*" condition variables), signaling does not cause the signaling thread to lose occupancy of the monitor. Instead the signaled threads are moved to the e queue. There is no need for the s queue.



A Mesa style monitor with two condition variables a and b

With nonblocking condition variables, the **signal** operation is often called **notify** — a terminology we will follow here. It is also common to provide a **notify all** operation that moves all threads waiting on a condition variable to the e queue.

The meaning of various operations are given here. (We assume that each operation runs in mutual exclusion to the others; thus restarted threads do not begin executing until the operation is complete.)

```

enter the monitor:
    enter the method
    if the monitor is locked
        add this thread to e
        block this thread
    else
        lock the monitor

```

```

leave the monitor:
  schedule
  return from the method

wait c :
  add this thread to c.q
  schedule
  block this thread

notify c :
  if there is a thread waiting on c.q
    select and remove one thread t from c.q
    (t is called "the notified thread")
    move t to e

notify all c :
  move all threads waiting on c.q to e

schedule :
  if there is a thread on e
    select and remove one thread from e and restart it
  else
    unlock the monitor

```

As a variation on this scheme, the notified thread may be moved to a queue called `w`, which has priority over `e`. See Howard<sup>[4]</sup> and Buhr *et al.*<sup>[5]</sup> for further discussion.

It is possible to associate an assertion  $P_c$  with each condition variable  $c$  such that  $P_c$  is sure to be true upon return from `wait c`. However, one must ensure that  $P_c$  is preserved from the time the **notifying** thread gives up occupancy until the notified thread is selected to re-enter the monitor. Between these times there could be activity by other occupants. Thus it is common for  $P_c$  to simply be *true*.

For this reason, it is usually necessary to enclose each **wait** operation in a loop like this

```
while not( P ) do wait c
```

where  $P$  is some condition stronger than  $P_c$ . The operations **notify**  $c$  and **notify all**  $c$  are treated as "hints" that  $P$  may be true for some waiting thread. Every iteration of such a loop past the first represents a lost notification; thus with nonblocking monitors, one must be careful to ensure that too many notifications can not be lost.

As an example of "hinting" consider a bank account in which a withdrawing thread will wait until the account has sufficient funds before proceeding

```

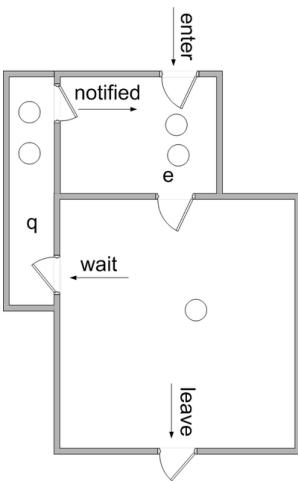
monitor class Account {
  private int balance := 0
  invariant balance >= 0
  private NonblockingCondition balanceMayBeBigEnough

  public method withdraw(int amount)
    precondition amount >= 0
  {
    while balance < amount do wait balanceMayBeBigEnough
    assert balance >= amount
    balance := balance - amount
  }

  public method deposit(int amount)
    precondition amount >= 0
  {
    balance := balance + amount
    notify all balanceMayBeBigEnough
  }
}
```

In this example, the condition being waited for is a function of the amount to be withdrawn, so it is impossible for a depositing thread to *know* that it made such a condition true. It makes sense in this case to allow each waiting thread into the monitor (one at a time) to check if its assertion is true.

## Implicit condition variable monitors



A Java style monitor

In the [Java](#) language, each object may be used as a monitor. Methods requiring mutual exclusion must be explicitly marked with the [synchronized](#) keyword. Blocks of code may also be marked by [synchronized](#).

Rather than having explicit condition variables, each monitor (i.e. object) is equipped with a single wait queue in addition to its entrance queue. All waiting is done on this single wait queue and all **notify** and **notifyAll** operations apply to this queue. This approach has been adopted in other languages, for example [C#](#).

### Implicit signaling

Another approach to signaling is to omit the **signal** operation. Whenever a thread leaves the monitor (by returning or waiting) the assertions of all waiting threads are evaluated until one is found to be true. In such a system, condition variables are not needed, but the assertions must be explicitly coded. The contract for wait is

```
wait P:  
  precondition I  
  modifies the state of the monitor  
  postcondition P and I
```

## History

Brinch Hansen and Hoare developed the monitor concept in the early 1970s, based on earlier ideas of their own and of [Edsger Dijkstra](#).<sup>[16]</sup> Brinch Hansen published the first monitor notation, adopting the [class](#) concept of [Simula 67](#),<sup>[11]</sup> and invented a queueing mechanism.<sup>[17]</sup> Hoare refined the rules of process resumption.<sup>[12]</sup> Brinch Hansen created the first implementation of monitors, in [Concurrent Pascal](#).<sup>[16]</sup> Hoare demonstrated their equivalence to [semaphores](#).

Monitors (and Concurrent Pascal) were soon used to structure process synchronization in the Solo operating system.<sup>[18][19]</sup>

Programming languages that have supported monitors include

- [Ada](#) since Ada 95 (as protected objects)
- [C#](#) (and other languages that use the [.NET Framework](#))
- [C++](#) since [C++11](#)
- [Concurrent Euclid](#)
- [Concurrent Pascal](#)
- [D](#)
- [Delphi](#) (Delphi 2009 and above, via `TObject.Monitor`)
- [Java](#) (via the `wait` and `notify` methods)
- [Mesa](#)
- [Modula-3](#)
- [Python](#) (via `threading.Condition` object)
- [Ruby](#)
- [Squeak](#) Smalltalk
- [Turing](#), [Turing+](#), and [Object-Oriented Turing](#)
- [μC++](#)

A number of libraries have been written that allow monitors to be constructed in languages that do not support them natively. When library calls are used, it is up to the programmer to explicitly mark the start and end of code executed with mutual exclusion. [Pthreads](#) is one such library.

## See also

- [Mutual exclusion](#)
- [Communicating sequential processes](#) - a later development of monitors by [C. A. R. Hoare](#)

## The Java™ Tutorials

**Trail:** Essential Classes

**Lesson:** Concurrency

**Section:** Synchronization

### Synchronized Methods

The Java programming language provides two basic synchronization idioms: *synchronized methods* and *synchronized statements*. The more complex of the two, synchronized statements, are described in the next section. This section is about synchronized methods.

To make a method synchronized, simply add the `synchronized` keyword to its declaration:

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

If `count` is an instance of `SynchronizedCounter`, then making these methods synchronized has two effects:

- First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
- Second, when a synchronized method exits, it automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Note that constructors cannot be synchronized — using the `synchronized` keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

---

**Warning:** When constructing an object that will be shared between threads, be very careful that a reference to the object does not "leak" prematurely. For example, suppose you want to maintain a `List` called `instances` containing every instance of class. You might be tempted to add the following line to your constructor:

```
instances.add(this);
```

But then other threads can use `instances` to access the object before construction of the object is complete.

---

Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through synchronized methods. (An important exception: `final` fields, which cannot be modified after the object is constructed, can be safely read through non-synchronized methods, once the object is constructed) This strategy is effective, but can present problems with [liveness](#), as we'll see later in this lesson.

**Previous page:** Memory Consistency Errors

**Next page:** Intrinsic Locks and Synchronization

## The Java™ Tutorials

**Trail:** Essential Classes

**Lesson:** Concurrency

**Section:** Synchronization

## Intrinsic Locks and Synchronization

Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock*. (The API specification often refers to this entity simply as a "monitor.") Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility.

Every object has an intrinsic lock associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to *acquire* the object's intrinsic lock before accessing them, and then *release* the intrinsic lock when it's done with them. A thread is said to *own* the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquisition of the same lock.

### Locks In Synchronized Methods

When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

You might wonder what happens when a static synchronized method is invoked, since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the `Class` object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.

### Synchronized Statements

Another way to create synchronized code is with *synchronized statements*. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

In this example, the `addName` method needs to synchronize changes to `lastName` and `nameCount`, but also needs to avoid synchronizing invocations of other objects' methods. (Invoking other objects' methods from synchronized code can create problems that are described in the section on [Liveness](#).) Without synchronized statements, there would have to be a separate, unsynchronized method for the sole purpose of invoking `nameList.add`.

Synchronized statements are also useful for improving concurrency with fine-grained synchronization. Suppose, for example, class `MsLunch` has two instance fields, `c1` and `c2`, that are never used together. All updates of these fields must be synchronized, but there's no reason to prevent an update of `c1` from being interleaved with an update of `c2` — and doing so reduces concurrency by creating unnecessary blocking. Instead of using synchronized methods or otherwise using the lock associated with `this`, we create two objects solely to provide locks.

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void incl() {
```

```
    synchronized(lock1) {
        c1++;
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

Use this idiom with extreme care. You must be absolutely sure that it really is safe to interleave access of the affected fields.

## Reentrant Synchronization

Recall that a thread cannot acquire a lock owned by another thread. But a thread *can* acquire a lock that it already owns. Allowing a thread to acquire the same lock more than once enables *reentrant synchronization*. This describes a situation where synchronized code, directly or indirectly, invokes a method that also contains synchronized code, and both sets of code use the same lock. Without reentrant synchronization, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block.

---

Your use of this page and all the material on pages under "The Java Tutorials" banner is subject to these [legal notices](#).

Problems with the examples? Try [Compiling and Running the Examples](#):  
[FAQs](#).

Copyright © 1995, 2015 Oracle and/or its affiliates. All rights reserved.

Complaints? Compliments? Suggestions? [Give us your feedback](#).

**Previous page:** Synchronized Methods

**Next page:** Atomic Access

# Measuring Operating Systems Performance

No matter how fast our hardware gets, the performance of our system always matters. Programmers tend to add complexity and sophistication to their systems to match any hardware performance improvements, so there is always a need to design software that achieves good performance on whatever hardware is available to us.

This instantly raises a question: what do we mean by performance? Most of us have an informal sense of what we mean. Perhaps we mean we don't want to wait for our commands to complete. Perhaps we mean we want to run complex calculations on very large data sets fast enough for the results to be useful. Perhaps we mean we want to get the most possible work through a given piece of hardware that we possibly can. Perhaps we mean we want to use as little space as possible on a storage device, or send as few bits across a network as possible. Maybe we care about how long a user has to wait before he starts to see some response from the system, but maybe we care about how long it takes for the entire job to complete.

As these potential answers to the performance question suggest, there's a lot more to understanding performance in a computer system than one might initially think. And there are yet more complexities in actually providing a valid answer to a particular performance question, such as "how many web requests per second can my server handle," or "will my VoIP call provide comprehensible speech to the listener if run it over a particular network," or "how many buffers should I allocate in my operating system to make sure that I/O is not delayed too much?"

## Metrics

Perhaps the first and most important step in determining the performance of your system is achieving clarity about what you really care about. If you don't know what you want to measure, you can be quite sure you won't successfully measure it. (If only mere clarity were enough to ensure that; but it is a vital first step.) What makes a difference in your system? What must go fast, and what is less important to speed up? In a word, what are your goals?

As a rule, if we care about system performance, we must quantify it. Saying your system is "fast" doesn't mean much. Saying it can perform a complex operation in 10 nanoseconds makes it a lot clearer what's going on. So any good performance investigation is targeted towards reducing the observed system behavior to some set of characteristic numbers. These numbers must have useful meanings relative to your goals. So latency is likely to be expressed in some unit of time, and throughput is likely to be expressed in some unit of work that is relevant to your system divided by some unit of time. The numbers we choose to characterize the performance of our systems are called *metrics*, and obviously their proper choice is of critical importance. Just because some quantity is measurable, however, does not make it a good metric. For instance, you can put your smart phone on a scale and measure its weight. That's a metric, but it won't tell you much about whether you can render video at a high enough frame rate to be

tolerable. On the other hand, it might be relevant in a usability study. The metrics you choose need to be relevant to your goals.

One more important point about metrics is that they must be practically measurable by you. That's limited by your ability to probe hardware and alter software. If you have a proprietary operating system whose source code you cannot see, you will have a hard time measuring what's happening inside it. You will probably need to be satisfied with observing what happens as you enter and leave the operating system, perhaps augmented by whatever information the system itself will divulge to you on request. Similarly, if you are measuring the performance of a web server that you do not run yourself, you probably can't run any code at all on that server, and you must choose metrics observable from places that you can reach. On the other hand, if you are measuring software running on top of an operating system, you might have more liberty to alter it for measurement purpose. Or you might not.

Another important point is that we are often using the system whose performance we wish to measure to actually capture our results. If the process of performing the measurements itself has a large effect on the performance, we may have obtained false readings for our metrics. Consider, for example, a measurement system that regularly writes records concerning file system behavior to the disk drive that stores that file system. Chances are that this experimental logging is competing, in a performance sense, with the actual behavior you are trying to measure. Instead of simply observing how long it would take to perform a set of reads and writes from a group of files in the file system, we are also moving the head to another place on the disk where we are storing our log. Those head movements would make the file system appear to be slower than it actually would be if your experimental framework were not logging data. If your experimental framework interferes with the processes you are trying to measure, you end up with a false reading for your metric that will not accurately describe the system's behavior when you aren't running your experiment.

## Complexity and the Role of Statistics in Measurement

Given that you've determined a good set of metrics, what next? By this point in your studies, you should be painfully aware of the complexities of large systems, particularly of operating systems. That complexity is going to have a big impact on your performance measurement studies. Consider what you already have learned about operating systems. They make heavy use of caching, with an expectation that some operations will be cache hits (probably fast) and some will be cache misses (probably slow). Interrupts will occur at unpredictable times, possibly resulting in altered performance of the code they break up. Different scheduling disciplines will insert varying delays into the performance of individual processes. Even if you have access to the operating system code, down at the hardware level there are caches and pipelines and other optimizations that you can't see at all, except that they produce varying performance results.

All of these characteristics mean that if you measure some metric on your system once, and then repeat that measurement again, you might see very different values. So what's

the right value? The one that makes your system look best? The worst one you've ever seen? Something else?

If you have taken a course in statistics, you may have a pretty good idea already about how to handle this issue. Don't measure the event of interest on your system only once. Measure it many times and treat the set of measurements as a probability distribution. You can then use the rich set of tools that this field of mathematics offers us to analyze your performance. Among the simplest of these tools are *mean*, *median*, and *mode*. The mean of a set of measurements is its average, the median is its middle point, and the mode is the most common value in the set. Means are useful for getting a single number that somehow captures something important about the entire data set. Medians are useful for getting a sense of where the measurements in a data set are "centered," in some sense. Mode is typically most useful when one value occurs far more often than any other, since then it can give you an idea of what result is most probable from a given data set. Mode tends to be less useful when the measurements are pretty evenly spread out.

It's often helpful to take a step further and work with quantities called indices of dispersion, which essentially describe how spread out a set of measurements are. *Range* is one such index of dispersion, describing the highest and lowest value observed. *Standard deviation* is another, describing the most commonly occurring range of values around the mean within the set. *Confidence intervals* describe the probability that a particular measurement is within a certain range.

In figure 1, we show a small set of sample data of latencies (in milliseconds) to access a disk block. Figure 2 shows various statistical properties of that data set. We calculated the mean by adding the 11 latencies and dividing by 11. We calculated the median by ordering the measurements and choosing the one in the middle. We calculated the mode by counting the number of times particular latencies occurred and choosing the one that occurred most often. To determine the range, we simply found the lowest measurement and the highest measurement. Standard deviation is calculated, typically, with a somewhat more complicated formula, but it is described in any detailed treatment of statistical properties.

Trial	Latency
1	27
2	31
3	28
4	26
5	30
6	35
7	31
8	29
9	32
10	25
11	33

Figure 1. Sample disk drive latencies

Mean	29.7
Median	30
Mode	31
Range	25-35
Standard deviation	3.07

Figure 2. Some statistical properties of the data in figure 1

These statistical properties are actually a bit more complex than they appear at first glance, and full understanding of their proper use is beyond the scope of either this chapter or an introductory course on operating systems. For example, there are actually several different ways to calculate means, and calculation of confidence intervals is typically based on assumptions about the probability distribution of the measurements. We will not further describe the many useful tools that the field of statistics offers, beyond recommending that those interested in understanding the performance of their systems really need good mastery of some of these tools.

You might wonder how many experimental runs you need to perform to fully capture the behavior of a system you're studying, given that there is possible statistical variation in the performance of each run. The field of statistics has useful tools for this purpose, as well, but they are beyond the scope of this chapter. In brief, the greater the degree of variability in what you're measuring, the more independent measurements you'll need to perform to get a pretty confident picture of its full behavior.

## Comparing Alternatives

Sometimes the purpose of your performance experiment is simply to characterize how a system performs according to one or more important metrics. In other cases, the system can be built, configured, or used in several different ways, and you want to know how well it performs in those varying situations. When this kind of comparative form of performance measurement is what you need, you have to take some care in how you go about doing it.

One issue is that possibly there are a large number of different options you could compare. There may be multiple dimensions in which you could examine the system. For example, you might want to know what would happen if you increased or decreased the amount of RAM allocated to buffer spaces, or what would happen if you used several different scheduling disciplines, or what would happen if you replaced the Ext3 file system with Btrfs. You might want to know what would happen if you made several of these changes in different ways.

Things you intentionally alter in performance experiments to determine which of several alternatives to use are called *factors*. In the paragraph above, the amount of buffer space allocated might be one factor, the scheduling discipline used might be a second factor, and the file system you chose might be a third factor. Factors can be set independently. For example, I might want to look at allocating 1Gbyte of buffer space, using Linux's Completely Fair Scheduling, and Btrfs, vs. allocating .5 Gbytes of buffer space, simple round robin, and Ext3. For each choice of settings of factors you want to investigate, you'll need to run some experiments. As mentioned earlier, you'll probably need to perform multiple runs to capture statistical variations.

Think about this a moment. If there are four different buffer sizes I want to investigate, three different scheduling disciplines, and two file systems, and I care about all possible combinations, how many sets of experiments am I going to have to run? In this relatively limited case, we're up to 24 sets of experiments, each of which will need to be run multiple times. One of the dangers of running performance experiments is getting too entranced with the many possibilities. If you're not careful, you might spend the rest of your life investigating some of these issues. Or, more likely, you'll get exhausted and give up before you get around to looking at the situations that are really important. A key element in successfully obtaining good performance results is striking a balance between investigating everything that's very important versus avoiding a combinatorial explosion of experimental settings. Maybe, on careful consideration, looking at two buffer sizes and ignoring one of the scheduling disciplines will still tell you what you need to know.

Another element controlling how much work you need to do is how many runs of each alternative you make to capture statistical variation against the number of alternatives you look at. Generally, the more you make, the higher your confidence in the statistical representativeness of your results (though there is a point of diminishing returns). Is it more important to consider more options or to have greater confidence in the performance of the options you do consider? That's for you to determine.

One can further generalize this issue to obtain perhaps the most important piece of advice we can give you before you undertake a set of performance measurements: **think first, measure second**. Determine exactly what you want to know and what you need to do to learn it, then design and perform experiments targeted at that required knowledge. Otherwise, you can spend arbitrarily large amounts of time hacking your way through the performance measurement jungle, possibly emerging at the end without having learned anything of importance.

There are other important issues in comparing the performance of various alternatives. One is to treat each alternative you measure fairly. Don't create experimental conditions that artificially favor one alternative over another. For example, in some cases things go slower the first time you do them than the second or later times. Caching often causes this effect. So if you ran alternative A first, then ran alternative B, B might appear to be faster not because it's better, but because it benefited from caching. Merely switching the order isn't enough, because now you've favored A. Running each alternative multiple times will help wash out these effects, particularly if you randomly intersperse the alternatives in different runs. (That's not always easy, if you need to perform heavyweight reconfiguration of the system to enable different alternatives.)

That is merely one example of being fair. Using the same settings for all runs (other than those that define the alternatives themselves) is another example. Resetting the system to the same state before starting each run is a third example. Even if caching is not involved, there may be some initial work that each system might need to do. If the system is not reset, the later runs can unfairly benefit from the configuration work done by earlier runs. Isolating the system from unrelated effects (such as updates to key pieces of software, external work loads not intended to be captured by your experiment, filling up the disk drive with your own experimental results, and so on) is also important.

## Sources of Performance Problems

We can experience poor performance in our systems for many reasons. Sometimes there is an overloaded resource, such as memory or network bandwidth or CPU cycles.

Sometimes a solution built into the software doesn't scale, so performance seems fine until the load on the system gets high. Then suddenly we fall off a performance cliff.

Sometimes we have built an inefficient implementation that puts in unnecessary overheads, such as copying a piece of data many times or making lots of calls to recursive functions to perform a tiny amount of real work.

The problem you have will affect how you go about looking for it. If you run performance measurements to uncover a scaling issue, a test that only runs a small number of iterations or on a small version of the problem may not produce useful results. If your problem is a bottleneck in your network, running tests that don't send messages across the network will never find it. If your problem is contention in scheduling, a performance experiment embedded in a single process won't provide much insight.

There's an obvious chicken-and-egg problem here. You might know performance is bad, but to run an experiment to determine exactly why, you need to know why performance is bad. Otherwise, you might waste your time running an irrelevant experiment that tells you nothing. So how do you get started?

In actuality, you often can get some clues without running any new experiments. The operating system will tell you, on request, how much memory is being used, CPU utilization, and many other statistics concerning the behavior of your processes. If there's plenty of free memory and the system is still running poorly, you'd probably waste your time building a performance experiment based on investigating the effects of varying memory usage. If there are rarely any ready processes waiting to run, scheduling is very likely not the source of your problem. Knowledge of the general architecture and expected behavior of the poorly performing system component can help, as well. If you know that the software that is running slow always works on pretty much the same quantity of data, it's probably not a scaling problem.

But usually these kinds of hints will only get you so far, and often they will provide indications, not actually identify the source of the problem. What then? Take the best knowledge you can easily obtain about your code and the observed performance problem and generate a hypothesis about why it's happening. Design an experiment that will test that hypothesis, proving or disproving it. Run the experiment and determine if your hypothesis is born out. If not, generate a new hypothesis (with, one would hope, a deeper knowledge base to work from than before) and try again. Obviously, there are elements of art, experience, and even luck in this process. But you've seen this kind of process before. It's much like finding a bug in a program, where you observe the erroneous behavior, make a hypothesis for its cause, add fixes or extract new information relevant to the hypothesis, and test it, until the bug is found and repaired. Generally, finding performance problems is harder than finding bugs, since it's harder to narrow the field in which you're searching, but the basic approach is similar.

Like finding functionality bugs, finding performance problems is a skill you are likely to develop with practice. You will come to learn the signals that point towards particular

classes of performance problems and develop instincts that lead you in the right direction more often than the wrong one. However, never mistake your experience or a good hunch for the results of an actual measurement program. Ultimately, the point of performance measurement is to reveal the actual truth of what's happening, and nothing short of measuring it is a substitute for that evidence.

## Workloads

One important aspect of running a performance experiment is the workload you use. In some cases, you are examining the performance of a particular program or operating system element, in which case you will tailor the workload to exercise that software. In many cases, you are looking for general performance in the face of typical overall system loads. In that situation, you need to generate a realistic workload for your system. Either way, somehow you must provide data sets, background activities, network traffic, and various other types of workload-related effects to test the performance.

There are different aspects of workloads that you need to think about when designing performance experiments. Your system is designed to do certain things: schedule processes, lay out a file system on a flash drive, respond to web requests, and so on. Obviously, one important aspect of the workload is the tasks you provide to your system directly related to its purpose: the set of processes to be scheduled, the files and file accesses to be handled, the web requests that clients generate. An equally important aspect of the workload, however, is based on the fact that operating systems are complex and involve simultaneous interactions of many different components that might affect each other in unpredictable ways. How your file system would perform if the only activity on the operating system was reads and writes to it is not the question you need to answer, as a rule. The important question is how it would perform in the face of all the other ordinary activities that the operating system would be doing in a real world setting. So your workload must also capture those background activities.

There are several different types of workloads typically used for performance measurement.

1. Traces – Take or otherwise obtain a detailed trace of the workload of the system in its ordinary activities. What such a trace consists of depends on the nature of what you are testing. For a web server, the trace is likely to be a set of web requests submitted to the server. For a mail server, it is likely to be a set of messages delivered to that server. For a file system, it might be a set of opens, reads, writes, and other file system operations. For an operating system component, it might be a set of applications that are run in a particular order with specified inputs. Whatever the trace might consist of, you capture it from the running system, saving it in a form that will allow you to recreate it in a faithful manner. Then, for each experimental run, you start from the beginning and run it to the end.

Traces have good and bad properties for performance experiments. A good property is realism, since they represent realistic activities that you would actually want your system to handle well. Another good property is reproducibility. The same trace can be replayed over and over, identically for each run. There is an

issue here if the performance of the system has an impact on what would have happened in the real system. For example, a trace of a network protocol that sends a message and receives an acknowledgement before sending the next message would have run differently if the acknowledgement had been produced in half the time, double the time, or at some other delay than it had been when the trace was gathered. If the system being tested is the one generating the acknowledgements, that can result in the replayed trace producing unrealistic results.

A disadvantage of a trace is that it is not easily reconfigurable. If your experiment needs to examine performance under controlled levels of workload, you might not be able to get a trace for each workload level you need. Merely running two copies of one trace in parallel might not realistically represent a true doubled load. Cutting out portions of a trace might not realistically represent a smaller workload, either. Scaling a trace up or down is usually hard. Another frequent disadvantage is availability. Good traces are not easy to come by, and if your system is not yet in production, you might be unable to gather your own. Except for freshly gathered traces of your own, most traces you can find will be somewhat (to very) old. Another disadvantage in some cases is that it might be difficult to gather the information needed to create the trace from the tools available to you. You might not be able to capture all the system calls applications perform, for instance. Also, any particular single trace might or might not represent the typical activity of the system. The moment at which it was gathered might have been unusual, compared to the ordinary activities of your system. Depending on exactly what you are tracing, there may be privacy implications to saving it in a trace. For certain kinds of system, such as those dealing with medical records, you may have legal obligations to handle some of the data in particular ways. Be aware of any such privacy problems before you store data for a trace.

2. Live workloads – Sometimes you can perform measurements on a working system as it goes about its normal activities. A production system can also be instrumented and data gathered as it does its work. Realism is a clear advantage here. Also, provided you can continue to do tests on the system indefinitely, with enough time you can capture a very wide range of real system behavior. You are likely to need to take little or no effort to establish realistic background loads, since they establish themselves, in essence.

This approach has its own disadvantages. One is lack of control, which manifests itself both in not being able to reproduce the behavior seen in previous tests, and in not being able to scale loads up and down as desired. Another is that your experimental framework usually needs to have minimal impact, both in performance and functionality, on the running system, since it is presumably more important to complete its live work than to gather your measurements. Unless this impact is essentially nil, you are not likely to be able to run the performance measurements for very long on a working system, since those tasked with getting it to do its job will not appreciate your experiments getting in the way. As with

traces, consider whether there are privacy implications to your observation of the live workload.

3. Standard benchmarks – These are either sets of programs or sets of data that are intended to drive performance experiments, typically on some particular thing, such as a file system, a database, a web server, or an intrusion detection system. They may have been derived from real traces at some point or they may be built from models of system behavior. They are typically designed to be usable by many developers, so it is often fairly easy to integrate them into your experiments, provided you are working in the same general framework they were designed for. (For example, a file system benchmark might generate Posix-compliant file operations, so any file system that is compatible with Posix can use it for testing.) They allow for easy comparison to other systems' performance, since the developers of those system can also run the same benchmark, or, indeed, you can yourself, if those other systems are also available for testing. A well-designed benchmark is likely to exercise a wide range of system behaviors, so the results you get from it may give you a fairly complete picture of your system's performance under different realistic conditions. Widely used benchmarks have been heavily studied themselves, and are unlikely to have many bugs, and likely to be relatively good representations of the kind of workload they are intended to mimic. Some benchmarks (though not all) are built to be inherently scalable, allowing you to adjust the workload up or down with little more than changing a line or two in a configuration file. Since benchmarks are artificial, there are usually no privacy implications to using them.

As you no doubt expect, though, standard benchmarks have their own set of disadvantages. First, there are a limited number of them available, and there might not be one suited for the system or situation you want to test. One aspect of this characteristic is that standard benchmarks might not include portions of the workload space that are unusual in general, but important for your case. Another aspect of this characteristic is that it's tempting to use a standard benchmark that isn't quite right for your situation just because it's easy to do so. Resist such temptations. Second, since developing a good benchmark is quite a lot of work, they tend to be used for a very long time, running the risk of representing archaic workloads that no longer match what would happen on a current system.

4. Simulated workloads – In this approach, you build models of the loads you are interested in, typically models instantiated in executable code. These models are usually parameterized, allowing them to be scaled up or down, to alter the mix of different elements of the load, and otherwise to create variations on the load. When testing a system's performance, one decides which parameter settings are most relevant and uses the simulated workload models with suitable settings. This approach has the advantage of being easily customized to many different scenarios and possibilities, since you need merely alter the model parameters accordingly. One important aspect of this flexibility is good handling of scaling, either up or down. Assuming that there is no true randomization in the models, they are infinitely repeatable, allowing you to perform directly comparable tests

of different system alternatives. As with standard benchmarks, the artificiality of simulated workloads has the benefit of avoiding privacy considerations.

However, the validity of the performance results you achieve is only as good as the quality of the models. It is not easy to produce good models of complex systems and phenomena, and one can easily overlook important features of real loads in building one's models. While parameters can be easily altered and scaled, even if the model was faithful to real load for some settings, it may prove unrealistic at others. It may also be unclear how to set the various parameters to produce simulated load that matches a particular real load. If the parameters are set incorrectly, one may get a very false picture of how a real system would behave in those situations.

## Common Mistakes in Performance Measurements

At the highest level, most mistakes in performance measurements can be attributed to insufficient thought by the experimenter. If you leap into a measurement program without giving some careful thought to what you are going to do, it may be hard to predict what problem you're going to encounter, but you've very likely to encounter some problem. Be clear on what issues you are investigating and what methods you are using to investigate them. Most frequently, you will benefit from writing down what you propose to do before you do it. That allows you to go back and check that you are continuing down the path you'd planned and hitting all the points you thought were important. It's OK to alter your plans as new information arises, but do so knowingly, not because you're blindly flailing around in a huge space of possible performance experiments.

At the next level of detail, problems tend to arise in areas like not measuring the right thing, not measuring accurately, not measuring in situations matching real world behavior, and not understanding what your measurements are telling you. These issues are so broad and have some many variants of the mistakes you can make, which are often quite specific to the system that you are measuring, that it is not necessarily helpful to pin down too many particular mistakes.

But there are certain more specific mistakes in measuring system performance that are sufficiently common that they are worth calling out in detail. We'll go through a few of these.

1. Measuring latency without considering utilization. Everything runs fast (or at least faster) on a lightly loaded system. Measuring the latency of an operation when absolutely nothing else is going on in the system is only worthwhile if the question to be answered is what is the fastest possible time in which it will complete. For anything else, one should measure the latency when the system has a characteristic background load. Most often, one should also examine the latency when the system is heavily loaded, as well, since that condition is likely to arise sooner or later in most systems.
2. Not reporting the variability of measurements. Sometimes this mistake is even more egregious, when a quantity is measured only once and that value reported as entire truth of the performance. Even if multiple measurements are taken,

- however, merely reporting the average of the values will often give a false impression of the performance observed. For most phenomena, one needs to understand the distribution of those values. Is it basically bi-modal? Is there one very common value and some outliers? Are the values uniformly spread across some range? What behavior you will observe in the real world and whether you will be happy with your system or miserable may depend on the answers to those questions, so a good performance experiment should offer you some insight into them.
3. Ignoring important special cases. This mistake comes in two varieties. In one, you ignore the fact that a few special cases distort the measurement, given you a false sense of what happens in the more general case. In the other, while you carefully measure the ordinary case, you fail to consider that there will be some special circumstances that are very important and that are likely to display different performance.

Perhaps the most common version of the first variety is ignoring startup effects. Computers make effective use of caching in many different ways. Programs loaded off disk may hang around in memory for a while in case they will be run again. Translations of DNS names to IP addresses are stored to avoid having to make expensive network requests multiple times. Hardware caches recently run instructions to avoid the cost of fetching them out of RAM when executing a loop. Caching is so ubiquitous and built into so many levels of a system that you are unlikely to predict all of its uses. That means you should regard the first few runs of a performance experiment as being potentially biased. They may have paid higher penalties than subsequent runs in order to warm up some caches. That does not necessarily mean you should discard them or disregard them, since, after all, every cache in a real system pays a performance penalty the first time the data is used, and that is a real element of system performance. But it does mean you should not compare different alternatives when one alternative has had the benefit of a warm cache while the other has not.

A similar problem can work in the opposite way. Sometimes we have a data structure of a limited size, and as long as we are working within that size, things go quickly. When we have more elements than the data structure can hold, performance degrades. For example, consider a hash table that uses chaining to handle collisions. If the table is relatively empty, every read will hit the element it was looking for immediately, and performance will be fast. When the table starts to fill up, some probes will need to follow a chain of entries to find the one they are looking for in that cache bucket; performance will slow down in some cases, while remaining fast for others. If the table is very full, performance will slow down for almost everything, since most probes will require searching a chain. File systems that use various kinds of indirect blocks are another example. Accesses to the first few blocks will avoid the indirect block and will be fast. Accesses further into a file will require indirect, doubly indirect, or triply indirect access, and will be slower, depending on the access pattern. Again, these are genuine performance effects, but only if you are trying to measure performance for conditions where they might occur.

The other variant is also important, because sometimes these genuine effects are critical to what you need to measure. If you only measure a file system's performance on short files, you may never learn that it is very slow once files exceed a certain size until, in production use, your system suddenly slows to a crawl. Special cases can be very important. Sometimes what you really need to know, for example, is how long servicing a web request will take under the worst circumstances likely to arise, or how your system will behave under extremely high load, or what will happen if a piece of hardware experiences partial failure. This issue returns to the point of understanding what you are measuring and why you are measuring it.

4. Ignoring the costs of your measurement program. In a few cases, you may be measuring a system using tools that are entirely external to the system and impose little or no load on that system. Sniffing traffic on a network is one example. More commonly, especially for operating system measurement, you are using your system to measure your system. You're not only sharing the processor, memory, network, and secondary storage devices with the system under study, but you're sharing some of the abstractions the operating system offers. For example, if you are logging information from your measurement code for later examination, you are probably exercising the file system. Does that matter? If what you're measuring is file system performance, it almost certainly does, and it might even if you are measuring something that does not have any obvious relationship to the file system, such as the scheduler or the memory manager. The file system is obviously not the only example. If you are running a separate process to perform your measurements, for example, it is competing for CPU and memory with the processes you are trying to measure.

Ideally, you want to avoid having your performance measurement program affect the behavior of the observed system at all. Often, this is impossible, in which case minimizing the impact and understanding it are the next best alternatives. There are useful techniques to minimize measurement costs. For example, instead of writing each observed measurement to disk (an expensive operation that interferes with other disk behavior), save it in a RAM buffer. Either write out the buffer to disk at the end of the experiment, if you can buffer that much data without affecting the experiment in other ways, or infrequently write it to disk if you can't keep a buffer that big. Keep your measurement code small and cheap. If feasible, bundle it into the process you are measuring, rather than running it as a separate process, unless you have reason to believe that it will cause less interference as a separate process. Avoid doing data analysis while gathering the data, since such analysis may be computationally expensive. Do it when you have finished the experiment and its calculations will not interfere with the experiment's timings. Start up anything computationally expensive related to your experiment before you start measuring the system you are investigating.

5. Losing your data. Never throw away experimental data, even if you think that you are finished with your experiment, nor even if you think the data in question was gathered in an erroneous way. Data has a way of proving useful for many

- purposes, but discarded data is never useful. Of course, you should particularly avoid carelessly losing data. One common beginner mistake is to inadvertently overwrite the data from a previous experiment with data being gathered for the next experiment. Also remember to label your data. Even if you have kept every byte of data you've gathered, if you can't tell which bytes are related to which parts of your experiment, the data is as good as lost. This advice is for the long term. Ideally, you should be able to go back and look at data you gathered twenty years ago. Maybe you will never look at a lot of the data you gathered in the past again, but probably you will eventually want to look at some of it, and it's hard to predict what's going to prove useful in the future. So save it all, if possible. It's also important to keep the metadata around, which in this case means information about how you set up and ran your experiments. Which version of the operating system was it that you used on that experiment you ran five years ago? Chances are you won't remember, so make sure it's written down somewhere you can find.
6. Valuing numbers over wisdom. Remember, the point of your performance experiment is not to obtain a set of numbers. It's to understand important performance characteristics of your system. The numbers are the means to an end, not themselves the end. Don't bother gathering numbers that are not going to lead to wisdom, and don't consider your task complete when you have the numbers in hand. You actually have the most important step still to go: using the numbers to understand your system performance and, if necessary, using them to guide redesign or reconfiguration of your system in ways that are likely to lead to better performance.

Unfortunately, it's hard to offer general advice on how to extract wisdom from sets of numbers. That's a task you will need to perform on a case-by-case basis. But do remember that performing that task is the goal, the entire point of running an elaborate performance experiment. Without the resulting wisdom, the work you did to get the numbers will be wasted.

# Load and Stress Testing

Mark Kampe

\$Id: loadstress.html 7 2007-08-26 19:52:08Z Mark \$

## 1. Introduction

Load and Stress Testing is very different from most other testing activities.

For most products, the vast majority of all tests exercise positive functional assertions (if situation x, the program will y). Such assertions may describe either positive (functionality) or negative (error handling) behavior. A typical suite is collection of test cases, each of which has the general form:

- establish conditions ( $c_1, c_2, \dots c_n$ )
- perform operations ( $o_1, o_2, \dots o_n$ )
- ascertain that assertions ( $a_1, a_2, \dots a_n$ ) are satisfied.

The art of creating such a test suite is being able to define a set of test cases that simultaneously:

- a. adequately exercises the program's capabilities
- b. adequately captures and verifies the program's behavior
- c. is small enough to be practically implementable

If all operations are performed, and all of the required assertions were satisfied, the program has passed the test. Such "functional validation" suites form the foundation for automated software testing, and are the primary basis for determining whether or not a product "works". As such, repeatability of results is considered to be very important.

Load and stress testing are quite different:

- the number of enumerated test cases is relatively small, and the particulars of each test case may be particularly important.
- we will run these test cases in pseudo-random orders for unspecified periods of time.
- we have no expectation that results will be repeatable (we are, in fact, depending on this).
- in many cases, there is no definitive pass indication. Rather, we can only say that the test has run for a period of time.
- we may not take the trouble to define a complete set of assertions to determine the correctness with which any particular operation has been performed. In some cases we may not even look at returned results.

This note is a brief introduction to the goals and methods of load and stress testing.

## 2. Load Testing

The initial (and perhaps still primary) purpose of load testing is to measure the ability of a system to provide service at a specified load level. A test harness issues requests to the component under test at a specified rate (the offered load), and collects performance data. The collected performance data depends on what is to be measured, but typically includes things like:

- response time for each request
- aggregate throughput
- CPU time and utilization

- disk I/O operations and utilization
- network packets and utilization

The resulting information can be used to:

- measure the system's speed and capacity
- analyze bottlenecks to enable improvements

The key ingredient in this process is a tool to generate the test traffic. Such tools are called **load generators**.

## 2.1 Load Generation

A load generator is a system that can generate test traffic, corresponding to a specified profile, at a calibrated rate. This traffic will be used to drive the system that is being measured. Such testing is normally performed on a "whole system", and is typically performed through the system's primary service interfaces:

- If the system to be tested is a network server, the load generator will pretend to be numerous clients, sending requests over a network.
- If the system to be tested is an application server, the load generator will create multiple test tasks to be run.
- If the system to be tested is an I/O device, the load generator will generate I/O requests.

In all cases, the test load is broadly characterized in terms of:

- request rate  
the number of operations per second
- request mix

Different types of clients use a system in different ways. A database might do a large number of small (and relatively random) disk reads and writes, while a streaming video server would do a much smaller number of huge contiguous transfers, all reads. If different types of requests exercise very different code paths, it is important that the load generator be able to accurately emulate all of the various types of clients.

- sequence fidelity

In many situations, it is sufficient to merely generate the right mix of read and write operations. In other cases it may be critical to simulate particular access patterns (timed sequences operations on related objects). In some situations it may be necessary to simulate realistic scenarios against pre-determined or random objects.

A good load generator will be tunable in terms of both the overall request rate and the mix of operations that is generated. Some load generators may merely generate random requests according to a distribution. Others may have rule grammars that describe typical scenarios, and use these to generate complex realistic usage scenarios.

Most such load generators are proprietary tools, developed and maintained by the organizations that build the products they measure. Some have been turned into products (or open source tools) and are widely used. Some have become so widely used that they have been adopted as "standard performance benchmarks".

## 2.2 Performance Measurement

There are a few typical ways to use a load generator for performance assessment:

1. Deliver requests at a specified rate and measure the response time.
2. Deliver requests at increasing rates until a maximum throughput is reached.
3. Deliver requests at a rate, and use this as a calibrated background for measuring the performance of other system services.

#### 4. Deliver requests at a rate, and use this as a test load for detailed studies performance bottlenecks.

In the first two usages, the load generator is a measurement tool. In the other usages, it provides a calibrated activity mix, to exercise the system.

### 2.3 Accelerated Aging

Many errors (e.g. memory leaks) can have trivial consequences, and only accumulate to measurable effects after long periods of time. Load generators can be used to create realistic traffic to simulate normal traffic for long periods of time. Alternatively, they can be cranked up to much higher rates in order to simulate accelerated aging. It is common for test plans to require products to undergo (at least) months of continuous load testing to look for the accumulation of such problems.

## 3. Stress Testing

Load generators are (fundamentally) intended to generate request sequences that are representative of what the measured system will experience in the field. Accelerated Aging uses cranked up load generators to simulate longer periods of use. If we go farther (into simulating scenarios far worse than anything that is ever expected to really happen) we enter the realm of stress testing.

Any error that results from a simple situation is likely to be easily recognized, tested, and (therefore) properly handled. In mature and well tested products, the residual errors tend to involve combinations of unlikely circumstances. These problems are much easier to find and fix in design review than they are to debug, but we still need to test for them. But how can we test for combinations of circumstances that we can't even enumerate?

The answer is random stress testing.

- use randomly generated complex usage scenarios, to increase the likelihood of encountering unlikely combinations of operations.
- deliberately generate large numbers conflicting requests (e.g. multiple clients trying to update the same file).
- introduce a wide range randomly generated errors, and simulated resource exhaustions, so that the system is continuously experiencing and recovering from errors.
- introduce wide swings in load, and regular overload situations.

Such testing takes situations that might normally occur only once or twice per year, and makes them occur (in combinations) hundreds of times per minute. Take a large number of such systems, and run them in this mode for several months. **This** will give us some serious confidence about the robustness and stability of our systems. Such testing is extremely demanding, and (in fact) very few software products receive (or survive) such testing. This is, however, typical methodology for mission critical and highly available products.

## 4. Conclusions

In the early stages of the product, most of the bugs are found by reviews and functional validation tests. These remain valuable (as regression tools) throughout the life of the product, but they are not actually expected to find many more bugs once they have initially succeeded.

Once a product has passed this "adolescent" stage, most of the bug reports result from new usage scenarios associated with adoption by real customers. There is a wider diversity of use here, and it may take a while to shake out all of these problems. But again, once the product has been brought up to the demands of every-day customer use, the number of bug reports resulting from this falls off sharply.

If we ignore new features (which are, in some sense, new software), the improvements in mature software tend to be in performance and robustness. The tools and techniques for finding functionality problems are neither designed nor adequate for driving improvements in these areas. Load and stress testing are very different from functional testing. The goals are different, the tools are different, the techniques are different, and the testing programs are different.

Functional quality starts with good design, and is then complemented by good review, and secured by a complete and well considered set of functional test suites. Performance and robustness also start with good design, are complemented by review, and secured by testing. Functionality tools and testing are, for the most part, done when the product ships. Load and stress testing will continue to be a critical element of product quality throughout its lifetime, and (unlike functional test cases) the load and stress testing tools may receive almost as much design and development attention as the tested product does.

In mature products, the difference between a good product and a great one is often found in the seriousness of the ongoing performance and robustness programs. **Performance** and **availability** don't just happen. They must be **earned**.

## Event-based Concurrency (Advanced)

Thus far, we've written about concurrency as if the only way to build concurrent applications is to use threads. Like many things in life, this is not completely true. Specifically, a different style of concurrent programming is often used in both GUI-based applications [O96] as well as some types of internet servers [PDZ99]. This style, known as **event-based concurrency**, has become popular in some modern systems, including server-side frameworks such as `node.js` [N13], but its roots are found in C/UNIX systems that we'll discuss below.

The problem that event-based concurrency addresses is two-fold. The first is that managing concurrency correctly in multi-threaded applications can be challenging; as we've discussed, missing locks, deadlock, and other nasty problems can arise. The second is that in a multi-threaded application, the developer has little or no control over what is scheduled at a given moment in time; rather, the programmer simply creates threads and then hopes that the underlying OS schedules them in a reasonable manner across available CPUs. Given the difficulty of building a general-purpose scheduler that works well in all cases for all workloads, sometimes the OS will schedule work in a manner that is less than optimal. The crux:

### THE CRUX:

#### HOW TO BUILD CONCURRENT SERVERS WITHOUT THREADS

How can we build a concurrent server without using threads, and thus retain control over concurrency as well as avoid some of the problems that seem to plague multi-threaded applications?

### 33.1 The Basic Idea: An Event Loop

The basic approach we'll use, as stated above, is called **event-based concurrency**. The approach is quite simple: you simply wait for something (i.e., an "event") to occur; when it does, you check what type of

event it is and do the small amount of work it requires (which may include issuing I/O requests, or scheduling other events for future handling, etc.). That's it!

Before getting into the details, let's first examine what a canonical event-based server looks like. Such applications are based around a simple construct known as the **event loop**. Pseudocode for an event loop looks like this:

```
while (1) {
    events = getEvents();
    for (e in events)
        processEvent(e);
}
```

It's really that simple. The main loop simply waits for something to do (by calling `getEvents()` in the code above) and then, for each event returned, processes them, one at a time; the code that processes each event is known as an **event handler**. Importantly, when a handler processes an event, it is the only activity taking place in the system; thus, deciding which event to handle next is equivalent to scheduling. This explicit control over scheduling is one of the fundamental advantages of the event-based approach.

But this discussion leaves us with a bigger question: how exactly does an event-based server determine which events are taking place, in particular with regards to network and disk I/O? Specifically, how can an event server tell if a message has arrived for it?

### 33.2 An Important API: `select()` (or `poll()`)

With that basic event loop in mind, we next must address the question of how to receive events. In most systems, a basic API is available, via either the `select()` or `poll()` system calls.

What these interfaces enable a program to do is simple: check whether there is any incoming I/O that should be attended to. For example, imagine that a network application (such as a web server) wishes to check whether any network packets have arrived, in order to service them. These system calls let you do exactly that.

Take `select()` for example. The manual page (on Mac OS X) describes the API in this manner:

```
int select(int nfds,
          fd_set *restrict readfds,
          fd_set *restrict writefds,
          fd_set *restrict errorfds,
          struct timeval *restrict timeout);
```

The actual description from the man page: `select()` examines the I/O descriptor sets whose addresses are passed in `readfds`, `writefds`, and `errorfds` to see if some of their descriptors are ready for reading, are ready for writing, or have

**ASIDE: BLOCKING VS. NON-BLOCKING INTERFACES**

Blocking (or **synchronous**) interfaces do all of their work before returning to the caller; non-blocking (or **asynchronous**) interfaces begin some work but return immediately, thus letting whatever work that needs to be done get done in the background.

The usual culprit in blocking calls is I/O of some kind. For example, if a call must read from disk in order to complete, it might block, waiting for the I/O request that has been sent to the disk to return.

Non-blocking interfaces can be used in any style of programming (e.g., with threads), but are essential in the event-based approach, as a call that blocks will halt all progress.

*an exceptional condition pending, respectively. The first nfds descriptors are checked in each set, i.e., the descriptors from 0 through nfds-1 in the descriptor sets are examined. On return, select() replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. select() returns the total number of ready descriptors in all the sets.*

A couple of points about `select()`. First, note that it lets you check whether descriptors can be *read* from as well as *written* to; the former lets a server determine that a new packet has arrived and is in need of processing, whereas the latter lets the service know when it is OK to reply (i.e., the outbound queue is not full).

Second, note the timeout argument. One common usage here is to set the timeout to `NULL`, which causes `select()` to block indefinitely, until some descriptor is ready. However, more robust servers will usually specify some kind of timeout; one common technique is to set the timeout to zero, and thus use the call to `select()` to return immediately.

The `poll()` system call is quite similar. See its manual page, or Stevens and Rago [SR05], for details.

Either way, these basic primitives give us a way to build a non-blocking event loop, which simply checks for incoming packets, reads from sockets with messages upon them, and replies as needed.

### 33.3 Using `select()`

To make this more concrete, let's examine how to use `select()` to see which network descriptors have incoming messages upon them. Figure 33.1 shows a simple example.

This code is actually fairly simple to understand. After some initialization, the server enters an infinite loop. Inside the loop, it uses the `FD_ZERO()` macro to first clear the set of file descriptors, and then uses `FD_SET()` to include all of the file descriptors from `minFD` to `maxFD` in the set. This set of descriptors might represent, for example, all of the net-

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/time.h>
4 #include <sys/types.h>
5 #include <unistd.h>
6
7 int main(void) {
8     // open and set up a bunch of sockets (not shown)
9     // main loop
10    while (1) {
11        // initialize the fd_set to all zero
12        fd_set readFDs;
13        FD_ZERO(&readFDs);
14
15        // now set the bits for the descriptors
16        // this server is interested in
17        // (for simplicity, all of them from min to max)
18        int fd;
19        for (fd = minFD; fd < maxFD; fd++)
20            FD_SET(fd, &readFDs);
21
22        // do the select
23        int rc = select(maxFD+1, &readFDs, NULL, NULL, NULL);
24
25        // check which actually have data using FD_ISSET()
26        int fd;
27        for (fd = minFD; fd < maxFD; fd++)
28            if (FD_ISSET(fd, &readFDs))
29                processFD(fd);
30    }
31 }
```

Figure 33.1: Simple Code Using `select()`

work sockets to which the server is paying attention. Finally, the server calls `select()` to see which of the connections have data available upon them. By then using `FD_ISSET()` in a loop, the event server can see which of the descriptors have data ready and process the incoming data.

Of course, a real server would be more complicated than this, and require logic to use when sending messages, issuing disk I/O, and many other details. For further information, see Stevens and Rago [SR05] for API information, or Pai et. al or Welsh et al. for a good overview of the general flow of event-based servers [PDZ99, WCB01].

### 33.4 Why Simpler? No Locks Needed

With a single CPU and an event-based application, the problems found in concurrent programs are no longer present. Specifically, because only one event is being handled at a time, there is no need to acquire or release locks; the event-based server cannot be interrupted by another thread because it is decidedly single threaded. Thus, concurrency bugs common in threaded programs do not manifest in the basic event-based approach.

**TIP: DON'T BLOCK IN EVENT-BASED SERVERS**

Event-based servers enable fine-grained control over scheduling of tasks. However, to maintain such control, no call that blocks the execution of the caller can ever be made; failing to obey this design tip will result in a blocked event-based server, frustrated clients, and serious questions as to whether you ever read this part of the book.

### 33.5 A Problem: Blocking System Calls

Thus far, event-based programming sounds great, right? You program a simple loop, and handle events as they arise. You don't even need to think about locking! But there is an issue: what if an event requires that you issue a system call that might block?

For example, imagine a request comes from a client into a server to read a file from disk and return its contents to the requesting client (much like a simple HTTP request). To service such a request, some event handler will eventually have to issue an `open()` system call to open the file, followed by a series of `read()` calls to read the file. When the file is read into memory, the server will likely start sending the results to the client.

Both the `open()` and `read()` calls may issue I/O requests to the storage system (when the needed metadata or data is not in memory already), and thus may take a long time to service. With a thread-based server, this is no issue: while the thread issuing the I/O request suspends (waiting for the I/O to complete), other threads can run, thus enabling the server to make progress. Indeed, this natural **overlap** of I/O and other computation is what makes thread-based programming quite natural and straightforward.

With an event-based approach, however, there are no other threads to run: just the main event loop. And this implies that if an event handler issues a call that blocks, the *entire* server will do just that: block until the call completes. When the event loop blocks, the system sits idle, and thus is a huge potential waste of resources. We thus have a rule that must be obeyed in event-based systems: no blocking calls are allowed.

### 33.6 A Solution: Asynchronous I/O

To overcome this limit, many modern operating systems have introduced new ways to issue I/O requests to the disk system, referred to generically as **asynchronous I/O**. These interfaces enable an application to issue an I/O request and return control immediately to the caller, before the I/O has completed; additional interfaces enable an application to determine whether various I/Os have completed.

For example, let us examine the interface provided on Mac OS X (other systems have similar APIs). The APIs revolve around a basic structure,

the `struct aiocb` or **AIO control block** in common terminology. A simplified version of the structure looks like this (see the manual pages for more information):

```
struct aiocb {
    int             aio_fildes;          /* File descriptor */
    off_t           aio_offset;         /* File offset */
    volatile void * aio_buf;            /* Location of buffer */
    size_t          aio_nbytes;         /* Length of transfer */
};
```

To issue an asynchronous read to a file, an application should first fill in this structure with the relevant information: the file descriptor of the file to be read (`aio_fildes`), the offset within the file (`aio_offset`) as well as the length of the request (`aio_nbytes`), and finally the target memory location into which the results of the read should be copied (`aio_buf`).

After this structure is filled in, the application must issue the asynchronous call to read the file; on Mac OS X, this API is simply the **asynchronous read API**:

```
int aio_read(struct aiocb *aiocbp);
```

This call tries to issue the I/O; if successful, it simply returns right away and the application (i.e., the event-based server) can continue with its work.

There is one last piece of the puzzle we must solve, however. How can we tell when an I/O is complete, and thus that the buffer (pointed to by `aio_buf`) now has the requested data within it?

One last API is needed. On Mac OS X, it is referred to (somewhat confusingly) as `aio_error()`. The API looks like this:

```
int aio_error(const struct aiocb *aiocbp);
```

This system call checks whether the request referred to by `aiocbp` has completed. If it has, the routine returns success (indicated by a zero); if not, `EINPROGRESS` is returned. Thus, for every outstanding asynchronous I/O, an application can periodically **poll** the system via a call to `aio_error()` to determine whether said I/O has yet completed.

One thing you might have noticed is that it is painful to check whether an I/O has completed; if a program has tens or hundreds of I/Os issued at a given point in time, should it simply keep checking each of them repeatedly, or wait a little while first, or ... ?

To remedy this issue, some systems provide an approach based on the **interrupt**. This method uses **UNIX signals** to inform applications when an asynchronous I/O completes, thus removing the need to repeatedly ask the system. This polling vs. interrupts issue is seen in devices too, as you will see (or already have seen) in the chapter on I/O devices.

**ASIDE: UNIX SIGNALS**

A huge and fascinating infrastructure known as **signals** is present in all modern UNIX variants. At its simplest, signals provide a way to communicate with a process. Specifically, a signal can be delivered to an application; doing so stops the application from whatever it is doing to run a **signal handler**, i.e., some code in the application to handle that signal. When finished, the process just resumes its previous behavior.

Each signal has a name, such as **HUP** (hang up), **INT** (interrupt), **SEGV** (segmentation violation), etc; see the manual page for details. Interestingly, sometimes it is the kernel itself that does the signaling. For example, when your program encounters a segmentation violation, the OS sends it a **SIGSEGV** (prepending **SIG** to signal names is common); if your program is configured to catch that signal, you can actually run some code in response to this erroneous program behavior (which can be useful for debugging). When a signal is sent to a process not configured to handle that signal, some default behavior is enacted; for **SEGV**, the process is killed.

Here is a simple program that goes into an infinite loop, but has first set up a signal handler to catch **SIGHUP**:

```
#include <stdio.h>
#include <signal.h>

void handle(int arg) {
    printf("stop wakin' me up...\n");
}

int main(int argc, char *argv[]) {
    signal(SIGHUP, handle);
    while (1)
        ; // doin' nothin' except catchin' some sigs
    return 0;
}
```

You can send signals to it with the **kill** command line tool (yes, this is an odd and aggressive name). Doing so will interrupt the main while loop in the program and run the handler code `handle()`:

```
prompt> ./main &
[3] 36705
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
```

There is a lot more to learn about signals, so much that a single page, much less a single chapter, does not nearly suffice. As always, there is one great source: Stevens and Rago [SR05]. Read more if interested.

In systems without asynchronous I/O, the pure event-based approach cannot be implemented. However, clever researchers have derived methods that work fairly well in their place. For example, Pai et al. [PDZ99] describe a hybrid approach in which events are used to process network packets, and a thread pool is used to manage outstanding I/Os. Read their paper for details.

### 33.7 Another Problem: State Management

Another issue with the event-based approach is that such code is generally more complicated to write than traditional thread-based code. The reason is as follows: when an event handler issues an asynchronous I/O, it must package up some program state for the next event handler to use when the I/O finally completes; this additional work is not needed in thread-based programs, as the state the program needs is on the stack of the thread. Adya et al. call this work **manual stack management**, and it is fundamental to event-based programming [A+02].

To make this point more concrete, let's look at a simple example in which a thread-based server needs to read from a file descriptor (`fd`) and, once complete, write the data that it read from the file to a network socket descriptor (`sd`). The code (ignoring error checking) looks like this:

```
int rc = read(fd, buffer, size);
rc = write(sd, buffer, size);
```

As you can see, in a multi-threaded program, doing this kind of work is trivial; when the `read()` finally returns, the code immediately knows which socket to write to because that information is on the stack of the thread (in the variable `sd`).

In an event-based system, life is not so easy. To perform the same task, we'd first issue the read asynchronously, using the AIO calls described above. Let's say we then periodically check for completion of the read using the `aio_error()` call; when that call informs us that the read is complete, how does the event-based server know what to do?

The solution, as described by Adya et al. [A+02], is to use an old programming language construct known as a **continuation** [FHK84]. Though it sounds complicated, the idea is rather simple: basically, record the needed information to finish processing this event in some data structure; when the event happens (i.e., when the disk I/O completes), look up the needed information and process the event.

In this specific case, the solution would be to record the socket descriptor (`sd`) in some kind of data structure (e.g., a hash table), indexed by the file descriptor (`fd`). When the disk I/O completes, the event handler would use the file descriptor to look up the continuation, which will return the value of the socket descriptor to the caller. At this point (finally), the server can then do the last bit of work to write the data to the socket.

### 33.8 What Is Still Difficult With Events

There are a few other difficulties with the event-based approach that we should mention. For example, when systems moved from a single CPU to multiple CPUs, some of the simplicity of the event-based approach disappeared. Specifically, in order to utilize more than one CPU, the event server has to run multiple event handlers in parallel; when doing so, the usual synchronization problems (e.g., critical sections) arise, and the usual solutions (e.g., locks) must be employed. Thus, on modern multicore systems, simple event handling without locks is no longer possible.

Another problem with the event-based approach is that it does not integrate well with certain kinds of systems activity, such as **paging**. For example, if an event-handler page faults, it will block, and thus the server will not make progress until the page fault completes. Even though the server has been structured to avoid *explicit* blocking, this type of *implicit* blocking due to page faults is hard to avoid and thus can lead to large performance problems when prevalent.

A third issue is that event-based code can be hard to manage over time, as the exact semantics of various routines changes [A+02]. For example, if a routine changes from non-blocking to blocking, the event handler that calls that routine must also change to accommodate its new nature, by ripping itself into two pieces. Because blocking is so disastrous for event-based servers, a programmer must always be on the lookout for such changes in the semantics of the APIs each event uses.

Finally, though asynchronous disk I/O is now possible on most platforms, it has taken a long time to get there [PDZ99], and it never quite integrates with asynchronous network I/O in as simple and uniform a manner as you might think. For example, while one would simply like to use the `select()` interface to manage all outstanding I/Os, usually some combination of `select()` for networking and the AIO calls for disk I/O are required.

### 33.9 Summary

We've presented a bare bones introduction to a different style of concurrency based on events. Event-based servers give control of scheduling to the application itself, but do so at some cost in complexity and difficulty of integration with other aspects of modern systems (e.g., paging). Because of these challenges, no single approach has emerged as best; thus, both threads and events are likely to persist as two different approaches to the same concurrency problem for many years to come. Read some research papers (e.g., [A+02, PDZ99, vB+03, WCB01]) or better yet, write some event-based code, to learn more.

## References

- [A+02] "Cooperative Task Management Without Manual Stack Management"  
 Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, John R. Douceur  
 USENIX ATC '02, Monterey, CA, June 2002  
*This gem of a paper is the first to clearly articulate some of the difficulties of event-based concurrency, and suggests some simple solutions, as well explores the even crazier idea of combining the two types of concurrency management into a single application!*
- [FHK84] "Programming With Continuations"  
 Daniel P. Friedman, Christopher T. Haynes, Eugene E. Kohlbecker  
 In Program Transformation and Programming Environments, Springer Verlag, 1984  
*The classic reference to this old idea from the world of programming languages. Now increasingly popular in some modern languages.*
- [N13] "Node.js Documentation"  
 By the folks who build node.js  
 Available: <http://nodejs.org/api/>  
*One of the many cool new frameworks that help you readily build web services and applications. Every modern systems hacker should be proficient in frameworks such as this one (and likely, more than one). Spend the time and do some development in one of these worlds and become an expert.*
- [O96] "Why Threads Are A Bad Idea (for most purposes)"  
 John Ousterhout  
 Invited Talk at USENIX '96, San Diego, CA, January 1996  
*A great talk about how threads aren't a great match for GUI-based applications (but the ideas are more general). Ousterhout formed many of these opinions while he was developing Tcl/Tk, a cool scripting language and toolkit that made it 100x easier to develop GUI-based applications than the state of the art at the time. While the Tk GUI toolkit lives on (in Python for example), Tcl seems to be slowly dying (unfortunately).*
- [PDZ99] "Flash: An Efficient and Portable Web Server"  
 Vivek S. Pai, Peter Druschel, Willy Zwaenepoel  
 USENIX '99, Monterey, CA, June 1999  
*A pioneering paper on how to structure web servers in the then-burgeoning Internet era. Read it to understand the basics as well as to see the authors' ideas on how to build hybrids when support for asynchronous I/O is lacking.*
- [SR05] "Advanced Programming in the UNIX Environment"  
 W. Richard Stevens and Stephen A. Rago  
 Addison-Wesley, 2005  
*Once again, we refer to the classic must-have-on-your-bookshelf book of UNIX systems programming. If there is some detail you need to know, it is in here.*
- [vB+03] "Capriccio: Scalable Threads for Internet Services"  
 Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, Eric Brewer  
 SOSP '03, Lake George, New York, October 2003  
*A paper about how to make threads work at extreme scale; a counter to all the event-based work ongoing at the time.*
- [WCB01] "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services"  
 Matt Welsh, David Culler, and Eric Brewer  
 SOSP '01, Banff, Canada, October 2001  
*A nice twist on event-based serving that combines threads, queues, and event-based handling into one streamlined whole. Some of these ideas have found their way into the infrastructures of companies such as Google, Amazon, and elsewhere.*

**Part III**

**Persistence**



## A Dialogue on Persistence

**Professor:** *And thus we reach the third of our four ... err... three pillars of operating systems: persistence.*

**Student:** *Did you say there were three pillars, or four? What is the fourth?*

**Professor:** *No. Just three, young student, just three. Trying to keep it simple here.*

**Student:** *OK, fine. But what is persistence, oh fine and noble professor?*

**Professor:** *Actually, you probably know what it means in the traditional sense, right? As the dictionary would say: "a firm or obstinate continuance in a course of action in spite of difficulty or opposition."*

**Student:** *It's kind of like taking your class: some obstinacy required.*

**Professor:** *Ha! Yes. But persistence here means something else. Let me explain. Imagine you are outside, in a field, and you pick a —*

**Student:** *(interrupting) I know! A peach! From a peach tree!*

**Professor:** *I was going to say apple, from an apple tree. Oh well; we'll do it your way, I guess.*

**Student:** *(stares blankly)*

**Professor:** *Anyhow, you pick a peach; in fact, you pick many many peaches, but you want to make them last for a long time. Winter is hard and cruel in Wisconsin, after all. What do you do?*

**Student:** *Well, I think there are some different things you can do. You can pickle it! Or bake a pie. Or make a jam of some kind. Lots of fun!*

**Professor:** *Fun? Well, maybe. Certainly, you have to do a lot more work to make the peach persist. And so it is with information as well; making information persist, despite computer crashes, disk failures, or power outages is a tough and interesting challenge.*

**Student:** *Nice segue; you're getting quite good at that.*

**Professor:** *Thanks! A professor can always use a few kind words, you know.*

**Student:** *I'll try to remember that. I guess it's time to stop talking peaches, and start talking computers?*

**Professor:** *Yes, it is that time...*

# Device Drivers: Classes and Services

## Introduction

Device drivers represent both:

generalizing abstractions

gathering a myriad of very different devices together and synthesizing a few general classes (e.g. disks, network interfaces, graphics adaptors) and standard models, behaviors and interfaces to be implemented by all drivers for a given class.

simplifying abstractions

providing an implementation of standard class interfaces while opaquely encapsulating the details of how to effectively and efficiently use a particular device.

For reasons of performance and control, Operating Systems tend not to be implemented in object oriented languages (does anybody remember JavaOS?). Yet despite being implemented in simpler languages (often C), Operating Systems, in device drivers, offer highly evolved examples of a different realization of class interfaces, derivation, and inheritance.

Whether we are talking about storage, networking, video, or human-interface, the number of available devices is huge and growing. The number and diversity of these devices creates tremendous demands for object oriented code reuse:

- We want the system to behave similarly, no matter what the underlying devices were being used to provide storage, networking, etc. To ensure this, we would like most of the higher level functionality to be implemented in common, higher level modules.
- We would like to minimize the cost of developing drivers to support new devices. This is most easily done if the majority of the functionality is implemented in common code that can be inherited by the individual drivers.
- As system functionality and performance are improved, we would like to ensure that those benefits accrue not only to new device drivers, but also to older device drivers.

These needs can be satisfied by implementing the higher level functionality (associated with each general class of device) in common code that uses per-device implementations of a standard sub-class driver to operate over a particular device. This requires:

- deriving device-driver sub-classes for each of the major classes of device.
- defining sub-class specific interfaces to be implemented by the drivers for all devices in each of those classes.
- creating per-device implementations of those standard sub-class interfaces.

## Major Driver Classes

In the earliest versions of Unix, all devices were divided into two fundamental classes, which are still present in all Unix derivatives:

block devices

These are random-access devices, addressable in fixed size (e.g. 512 byte, 4K byte) blocks. Their drivers implement a *request* method to enqueue asynchronous DMA requests. The request descriptor included information about the desired operation (e.g. byte count, target device, disk address and in-memory buffer address) as well as completion information (how much data was transferred, error indications) and a condition variable the requestor could use to await the eventual completion of the request.

A read or write request could be issued for any number of blocks, but in most cases a large request would be broken into multiple single-block requests, each of which would be passed, block at a time, through the system buffer cache. For this reason block device drivers also implement a *fsync* method to flush out any buffered writes.

## character devices

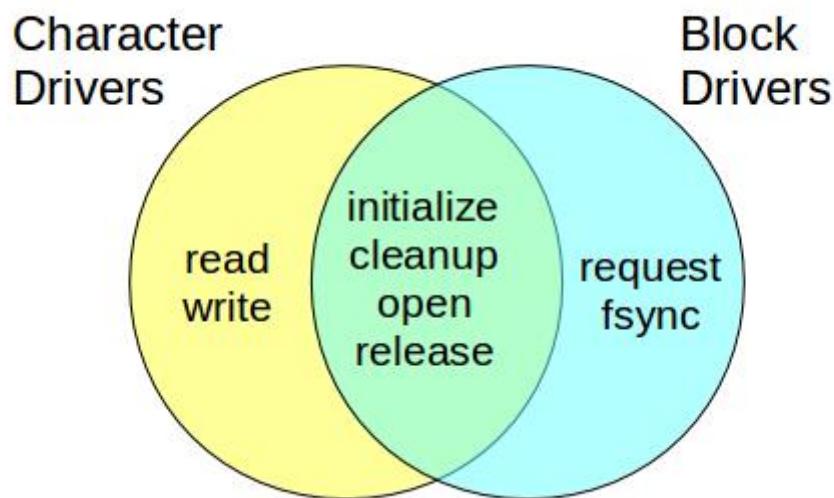
These devices may be sequential access, or may be byte-addressable. They support the standard synchronous *read(2)*, *write(2)* and (indirectly) *seek(2)* operations.

For devices that supported DMA, read and write operations were expected to be done as a single (potentially very large) DMA transfer between the device and the buffers in user address space.

A key point here is that, even in the oldest Unix systems, device drivers were divided into distinct classes (implementing different interfaces) based on the needs of distinct classes of clients:

- Block devices were designed to be used, within the operating system, by file systems, to access disks. Forcing all I/O to go through the system buffer cache is almost surely the right thing to do with file system I/O.
- Character devices were designed to be used directly by applications. The potential for large DMA transfers directly between the device and user-space buffers meant that character (or *raw*) I/O operations might be much more efficient than the corresponding requests to a block device.

These two major classes of device were not mutually exclusive. A single driver could export both block and character interfaces. A file system would be mounted on top of the block device, while back-up and integrity-checking software might access the disk through its (potentially much more efficient) character device\*. All device drivers support *initialize* and *cleanup* methods (for dynamic module loading and unloading), *open* and *release* methods (roughly corresponding to the *open(2)* and *close(2)* system calls), and an optional catch-all *ioctl(2)* method.

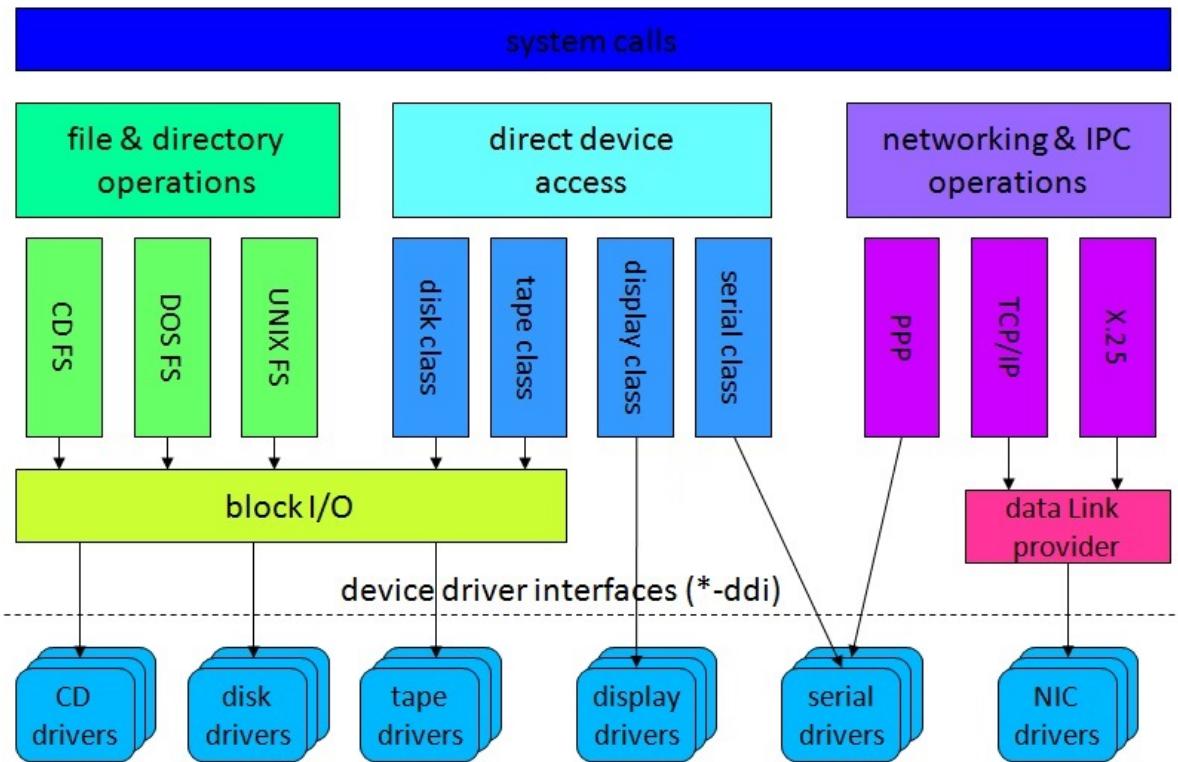


\*In more contemporary systems, it is possible for a client to specify that block I/O should not be passed through the system buffer cache.

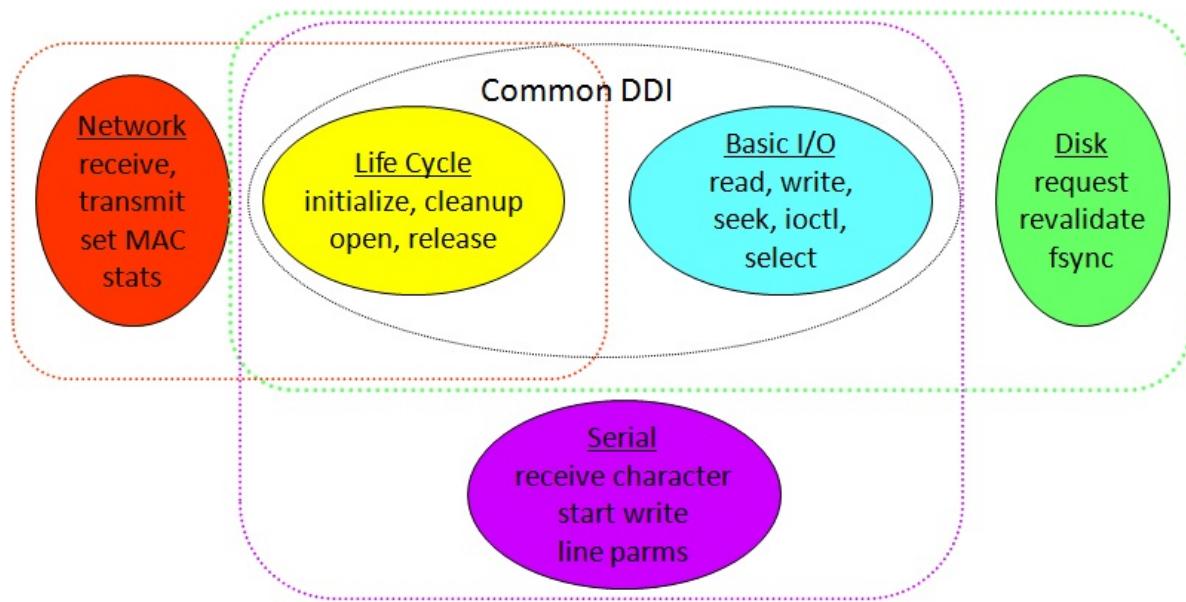
# Driver sub-classes

Given the fundamental importance of file systems and disks to operating systems, it is not surprising that block device drivers would have been singled out as a special sub-class in even the earliest versions of Unix. But as system functionality evolved, the operating system began to implement higher level services for other sub-classes of devices:

- input editing and output translation for terminals and virtual terminals
- address binding and packet sending/receipt for network interfaces
- display mapping and window management for graphics adaptors
- character-set mapping for keyboards
- cursor positioning for pointing devices
- sample mixing, volume and equalization for sound devices



And as each of these sub-systems evolved, new device driver methods\* were defined to enable more effective communication between the higher level frameworks and the lower level device drivers in each sub-class. Each of these sub-class specific interfaces is referred to as a *Device Driver Interface* (DDI).



\*It should be noted that in some cases, the higher frameworks have been implemented in user-mode, so that some of the new interfaces have been specified as behavior rather than new methods.

The rewards for this structure are:

- Sub-class drivers become easier to implement because so much of the important functionality is implemented in higher level software.
- The system behaves identically over a wide range of different devices.
- Most functionality enhancements will be in the higher level code, and so should automatically work on all devices within that sub-class.

But the price for these rewards is that all device drivers must implement exactly the same interfaces:

- if a driver does not (correctly) implement the standard interfaces for its device sub-class, it will not work with the higher level software.
- if a driver implements additional functionality (not defined in the standard interfaces for its device sub-class), those features will not be exploited by the standard higher level software.

## Services for Device Drivers

It is nearly impossible to implement a completely self-contained device driver. Most device drivers are likely to require a range of resources and services from the operating system:

- dynamic memory allocation
- I/O and bus resource allocation and management
- condition variable operations (wait and signal)
- mutual exclusion
- control of, and being called to service interrupts
- DMA target pin/release, scatter/gather map management
- configuration/registry services

The collection of services, exposed by the operating system for use by device drivers is sometimes referred to as the *Driver-Kernel Interface* (DKI). Interface stability for DKI functions is every bit as important as it is for the DDI entry points. If an operating system eliminates or incompatibly changes a DKI function, device drivers that depend on that function may cease working. The requirement to maintain stable DKI entry points may greatly constrain our ability to evolve our operating system implementation. Similar issues arrise for other classes of dynamically loadable kernel modules (such as network protocols and file systems).

## Conclusion

Device drivers demonstrate an evolution from a basic super-class (character devices) into an ever-expanding hierarchy of derived sub-classes. But unlike traditional class derivation, where sub-class implementations inherit most of their implemntation from their parent, we see a different sort of inheritance. While each new sub-class and instance is likely to be a new implementation, what they inherit is pre-existing higher level frameworks that do do most of their work for them.

## I/O Devices

Before delving into the main content of this part of the book (on persistence), we first introduce the concept of an **input/output (I/O) device** and show how the operating system might interact with such an entity. I/O is quite critical to computer systems, of course; imagine a program without any input (it produces the same result each time); now imagine a program with no output (what was the purpose of it running?). Clearly, for computer systems to be interesting, both input and output are required. And thus, our general problem:

### CRUX: HOW TO INTEGRATE I/O INTO SYSTEMS

How should I/O be integrated into systems? What are the general mechanisms? How can we make them efficient?

### 36.1 System Architecture

To begin our discussion, let's look at the structure of a typical system (Figure 36.1). The picture shows a single CPU attached to the main memory of the system via some kind of **memory bus** or interconnect. Some devices are connected to the system via a general **I/O bus**, which in many modern systems would be PCI (or one of its many derivatives); graphics and some other higher-performance I/O devices might be found here. Finally, even lower down are one or more of what we call a **peripheral bus**, such as **SCSI**, **SATA**, or **USB**. These connect the slowest devices to the system, including **disks**, **mice**, and other similar components.

One question you might ask is: why do we need a hierarchical structure like this? Put simply: physics, and cost. The faster a bus is, the shorter it must be; thus, a high-performance memory bus does not have much room to plug devices and such into it. In addition, engineering a bus for high performance is quite costly. Thus, system designers have adopted this hierarchical approach, where components that demand high performance (such as the graphics card) are nearer the CPU. Lower per-

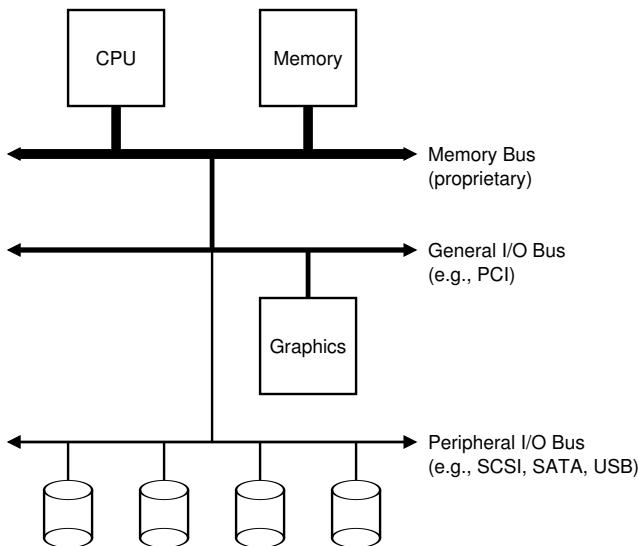


Figure 36.1: **Prototypical System Architecture**

formance components are further away. The benefits of placing disks and other slow devices on a peripheral bus are manifold; in particular, you can place a large number of devices on it.

## 36.2 A Canonical Device

Let us now look at a canonical device (not a real one), and use this device to drive our understanding of some of the machinery required to make device interaction efficient. From Figure 36.2, we can see that a device has two important components. The first is the hardware **interface** it presents to the rest of the system. Just like a piece of software, hardware must also present some kind of interface that allows the system software to control its operation. Thus, all devices have some specified interface and protocol for typical interaction.

The second part of any device is its **internal structure**. This part of the device is implementation specific and is responsible for implementing the abstraction the device presents to the system. Very simple devices will have one or a few hardware chips to implement their functionality; more complex devices will include a simple CPU, some general purpose memory, and other device-specific chips to get their job done. For example, modern RAID controllers might consist of hundreds of thousands of lines of **firmware** (i.e., software within a hardware device) to implement its functionality.

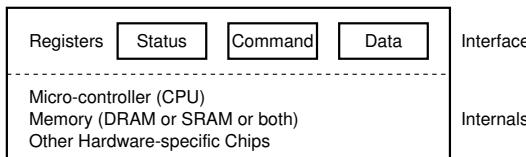


Figure 36.2: A Canonical Device

### 36.3 The Canonical Protocol

In the picture above, the (simplified) device interface is comprised of three registers: a **status** register, which can be read to see the current status of the device; a **command** register, to tell the device to perform a certain task; and a **data** register to pass data to the device, or get data from the device. By reading and writing these registers, the operating system can control device behavior.

Let us now describe a typical interaction that the OS might have with the device in order to get the device to do something on its behalf. The protocol is as follows:

```

While (STATUS == BUSY)
    ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    (Doing so starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device is done with your request

```

The protocol has four steps. In the first, the OS waits until the device is ready to receive a command by repeatedly reading the status register; we call this **polling** the device (basically, just asking it what is going on). Second, the OS sends some data down to the data register; one can imagine that if this were a disk, for example, that multiple writes would need to take place to transfer a disk block (say 4KB) to the device. When the main CPU is involved with the data movement (as in this example protocol), we refer to it as **programmed I/O (PIO)**. Third, the OS writes a command to the command register; doing so implicitly lets the device know that both the data is present and that it should begin working on the command. Finally, the OS waits for the device to finish by again polling it in a loop, waiting to see if it is finished (it may then get an error code to indicate success or failure).

This basic protocol has the positive aspect of being simple and working. However, there are some inefficiencies and inconveniences involved. The first problem you might notice in the protocol is that polling seems inefficient; specifically, it wastes a great deal of CPU time just waiting for the (potentially slow) device to complete its activity, instead of switching to another ready process and thus better utilizing the CPU.

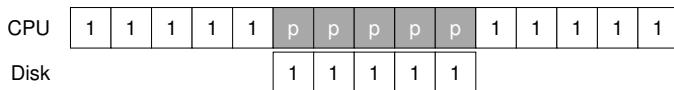
### THE CRUX: HOW TO AVOID THE COSTS OF POLLING

How can the OS check device status without frequent polling, and thus lower the CPU overhead required to manage the device?

## 36.4 Lowering CPU Overhead With Interrupts

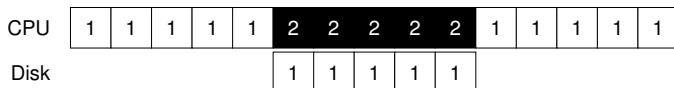
The invention that many engineers came upon years ago to improve this interaction is something we've seen already: the **interrupt**. Instead of polling the device repeatedly, the OS can issue a request, put the calling process to sleep, and context switch to another task. When the device is finally finished with the operation, it will raise a hardware interrupt, causing the CPU to jump into the OS at a pre-determined **interrupt service routine (ISR)** or more simply an **interrupt handler**. The handler is just a piece of operating system code that will finish the request (for example, by reading data and perhaps an error code from the device) and wake the process waiting for the I/O, which can then proceed as desired.

Interrupts thus allow for **overlap** of computation and I/O, which is key for improved utilization. This timeline shows the problem:



In the diagram, Process 1 runs on the CPU for some time (indicated by a repeated 1 on the CPU line), and then issues an I/O request to the disk to read some data. Without interrupts, the system simply spins, polling the status of the device repeatedly until the I/O is complete (indicated by a p). The disk services the request and finally Process 1 can run again.

If instead we utilize interrupts and allow for overlap, the OS can do something else while waiting for the disk:



In this example, the OS runs Process 2 on the CPU while the disk services Process 1's request. When the disk request is finished, an interrupt occurs, and the OS wakes up Process 1 and runs it again. Thus, *both* the CPU and the disk are properly utilized during the middle stretch of time.

Note that using interrupts is not *always* the best solution. For example, imagine a device that performs its tasks very quickly: the first poll usually finds the device to be done with task. Using an interrupt in this case will actually *slow down* the system: switching to another process, handling the interrupt, and switching back to the issuing process is expensive. Thus, if a device is fast, it may be best to poll; if it is slow, interrupts, which allow

**TIP: INTERRUPTS NOT ALWAYS BETTER THAN PIO**

Although interrupts allow for overlap of computation and I/O, they only really make sense for slow devices. Otherwise, the cost of interrupt handling and context switching may outweigh the benefits interrupts provide. There are also cases where a flood of interrupts may overload a system and lead it to livelock [MR96]; in such cases, polling provides more control to the OS in its scheduling and thus is again useful.

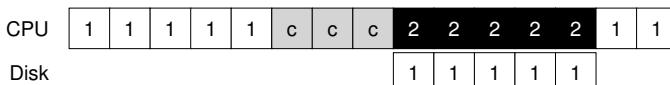
overlap, are best. If the speed of the device is not known, or sometimes fast and sometimes slow, it may be best to use a **hybrid** that polls for a little while and then, if the device is not yet finished, uses interrupts. This **two-phased** approach may achieve the best of both worlds.

Another reason not to use interrupts arises in networks [MR96]. When a huge stream of incoming packets each generate an interrupt, it is possible for the OS to **livelock**, that is, find itself only processing interrupts and never allowing a user-level process to run and actually service the requests. For example, imagine a web server that suddenly experiences a high load due to the “slashdot effect”. In this case, it is better to occasionally use polling to better control what is happening in the system and allow the web server to service some requests before going back to the device to check for more packet arrivals.

Another interrupt-based optimization is **coalescing**. In such a setup, a device which needs to raise an interrupt first waits for a bit before delivering the interrupt to the CPU. While waiting, other requests may soon complete, and thus multiple interrupts can be coalesced into a single interrupt delivery, thus lowering the overhead of interrupt processing. Of course, waiting too long will increase the latency of a request, a common trade-off in systems. See Ahmad et al. [A+11] for an excellent summary.

### 36.5 More Efficient Data Movement With DMA

Unfortunately, there is one other aspect of our canonical protocol that requires our attention. In particular, when using programmed I/O (PIO) to transfer a large chunk of data to a device, the CPU is once again overburdened with a rather trivial task, and thus wastes a lot of time and effort that could better be spent running other processes. This timeline illustrates the problem:



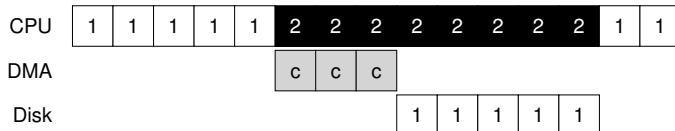
In the timeline, Process 1 is running and then wishes to write some data to the disk. It then initiates the I/O, which must copy the data from memory to the device explicitly, one word at a time (marked c in the diagram). When the copy is complete, the I/O begins on the disk and the CPU can finally be used for something else.

### THE CRUX: HOW TO LOWER PIO OVERHEADS

With PIO, the CPU spends too much time moving data to and from devices by hand. How can we offload this work and thus allow the CPU to be more effectively utilized?

The solution to this problem is something we refer to as **Direct Memory Access (DMA)**. A DMA engine is essentially a very specific device within a system that can orchestrate transfers between devices and main memory without much CPU intervention.

DMA works as follows. To transfer data to the device, for example, the OS would program the DMA engine by telling it where the data lives in memory, how much data to copy, and which device to send it to. At that point, the OS is done with the transfer and can proceed with other work. When the DMA is complete, the DMA controller raises an interrupt, and the OS thus knows the transfer is complete. The revised timeline:



From the timeline, you can see that the copying of data is now handled by the DMA controller. Because the CPU is free during that time, the OS can do something else, here choosing to run Process 2. Process 2 thus gets to use more CPU before Process 1 runs again.

## 36.6 Methods Of Device Interaction

Now that we have some sense of the efficiency issues involved with performing I/O, there are a few other problems we need to handle to incorporate devices into modern systems. One problem you may have noticed thus far: we have not really said anything about how the OS actually communicates with the device! Thus, the problem:

### THE CRUX: HOW TO COMMUNICATE WITH DEVICES

How should the hardware communicate with a device? Should there be explicit instructions? Or are there other ways to do it?

Over time, two primary methods of device communication have developed. The first, oldest method (used by IBM mainframes for many years) is to have explicit **I/O instructions**. These instructions specify a way for the OS to send data to specific device registers and thus allow the construction of the protocols described above.

For example, on x86, the `in` and `out` instructions can be used to communicate with devices. For example, to send data to a device, the caller specifies a register with the data in it, and a specific *port* which names the device. Executing the instruction leads to the desired behavior.

Such instructions are usually **privileged**. The OS controls devices, and the OS thus is the only entity allowed to directly communicate with them. Imagine if any program could read or write the disk, for example: total chaos (as always), as any user program could use such a loophole to gain complete control over the machine.

The second method to interact with devices is known as **memory-mapped I/O**. With this approach, the hardware makes device registers available as if they were memory locations. To access a particular register, the OS issues a load (to read) or store (to write) the address; the hardware then routes the load/store to the device instead of main memory.

There is not some great advantage to one approach or the other. The memory-mapped approach is nice in that no new instructions are needed to support it, but both approaches are still in use today.

## 36.7 Fitting Into The OS: The Device Driver

One final problem we will discuss: how to fit devices, each of which have very specific interfaces, into the OS, which we would like to keep as general as possible. For example, consider a file system. We'd like to build a file system that worked on top of SCSI disks, IDE disks, USB keychain drives, and so forth, and we'd like the file system to be relatively oblivious to all of the details of how to issue a read or write request to these difference types of drives. Thus, our problem:

### THE CRUX: HOW TO BUILD A DEVICE-NEUTRAL OS

How can we keep most of the OS device-neutral, thus hiding the details of device interactions from major OS subsystems?

The problem is solved through the age-old technique of **abstraction**. At the lowest level, a piece of software in the OS must know in detail how a device works. We call this piece of software a **device driver**, and any specifics of device interaction are encapsulated within.

Let us see how this abstraction might help OS design and implementation by examining the Linux file system software stack. Figure 36.3 is a rough and approximate depiction of the Linux software organization. As you can see from the diagram, a file system (and certainly, an application above) is completely oblivious to the specifics of which disk class it is using; it simply issues block read and write requests to the generic block layer, which routes them to the appropriate device driver, which handles the details of issuing the specific request. Although simplified, the diagram shows how such detail can be hidden from most of the OS.

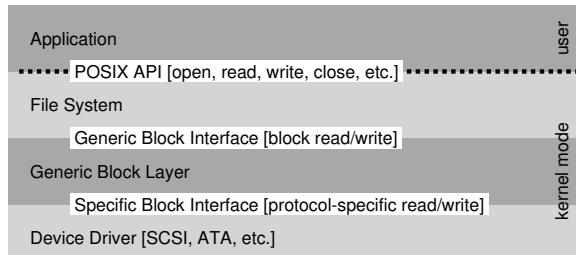


Figure 36.3: The File System Stack

Note that such encapsulation can have its downside as well. For example, if there is a device that has many special capabilities, but has to present a generic interface to the rest of the kernel, those special capabilities will go unused. This situation arises, for example, in Linux with SCSI devices, which have very rich error reporting; because other block devices (e.g., ATA/IDE) have much simpler error handling, all that higher levels of software ever receive is a generic `EIO` (generic IO error) error code; any extra detail that SCSI may have provided is thus lost to the file system [G08].

Interestingly, because device drivers are needed for any device you might plug into your system, over time they have come to represent a huge percentage of kernel code. Studies of the Linux kernel reveal that over 70% of OS code is found in device drivers [C01]; for Windows-based systems, it is likely quite high as well. Thus, when people tell you that the OS has millions of lines of code, what they are really saying is that the OS has millions of lines of device-driver code. Of course, for any given installation, most of that code may not be active (i.e., only a few devices are connected to the system at a time). Perhaps more depressingly, as drivers are often written by “amateurs” (instead of full-time kernel developers), they tend to have many more bugs and thus are a primary contributor to kernel crashes [S03].

## 36.8 Case Study: A Simple IDE Disk Driver

To dig a little deeper here, let’s take a quick look at an actual device: an IDE disk drive [L94]. We summarize the protocol as described in this reference [W10]; we’ll also peek at the xv6 source code for a simple example of a working IDE driver [CK+08].

An IDE disk presents a simple interface to the system, consisting of four types of register: control, command block, status, and error. These registers are available by reading or writing to specific “I/O addresses” (such as `0x3F6` below) using (on x86) the `in` and `out` I/O instructions.

```

Control Register:
Address 0x3F6 = 0x08 (0000 1RE0): R=reset, E=0 means "enable interrupt"

Command Block Registers:
Address 0x1F0 = Data Port
Address 0x1F1 = Error
Address 0x1F2 = Sector Count
Address 0x1F3 = LBA low byte
Address 0x1F4 = LBA mid byte
Address 0x1F5 = LBA hi byte
Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive
Address 0x1F7 = Command/status

Status Register (Address 0x1F7):
    7      6      5      4      3      2      1      0
    BUSY   READY  FAULT SEEK  DRQ   CORR IDDEX ERROR

Error Register (Address 0x1F1): (check when Status ERROR==1)
    7      6      5      4      3      2      1      0
    BBK    UNC    MC     IDNF   MCR   ABRT  TONF AMNF

BBK = Bad Block
UNC = Uncorrectable data error
MC = Media Changed
IDNF = ID mark Not Found
MCR = Media Change Requested
ABRT = Command aborted
TONF = Track 0 Not Found
AMNF = Address Mark Not Found

```

Figure 36.4: The IDE Interface

The basic protocol to interact with the device is as follows, assuming it has already been initialized.

- **Wait for drive to be ready.** Read Status Register (0x1F7) until drive is not busy and READY.
- **Write parameters to command registers.** Write the sector count, logical block address (LBA) of the sectors to be accessed, and drive number (master=0x00 or slave=0x10, as IDE permits just two drives) to command registers (0x1F2-0x1F6).
- **Start the I/O.** by issuing read/write to command register. Write READ—WRITE command to command register (0x1F7).
- **Data transfer (for writes):** Wait until drive status is READY and DRQ (drive request for data); write data to data port.
- **Handle interrupts.** In the simplest case, handle an interrupt for each sector transferred; more complex approaches allow batching and thus one final interrupt when the entire transfer is complete.
- **Error handling.** After each operation, read the status register. If the ERROR bit is on, read the error register for details.

Most of this protocol is found in the xv6 IDE driver (Figure 36.5), which (after initialization) works through four primary functions. The first is `ide_rw()`, which queues a request (if there are others pending), or issues it directly to the disk (via `ide_start_request()`); in either

```

static int ide_wait_ready() {
    while (((int r = inb(0x1f7)) & IDE_BSY) || !(r & IDE_DRDY))
        ;                                     // loop until drive isn't busy
}

static void ide_start_request(struct buf *b) {
    ide_wait_ready();
    outb(0x3f6, 0);                      // generate interrupt
    outb(0x1f2, 1);                      // how many sectors?
    outb(0x1f3, b->sector & 0xff);      // LBA goes here ...
    outb(0x1f4, (b->sector >> 8) & 0xff); // ... and here
    outb(0x1f5, (b->sector >> 16) & 0xff); // ... and here!
    outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
    if(b->flags & B_DIRTY) {
        outb(0x1f7, IDE_CMD_WRITE);       // this is a WRITE
        outsl(0x1f0, b->data, 512/4);   // transfer data too!
    } else {
        outb(0x1f7, IDE_CMD_READ);       // this is a READ (no data)
    }
}

void ide_rw(struct buf *b) {
    acquire(&ide_lock);
    for (struct buf **pp = &ide_queue; *pp; pp=&(*pp)->qnext)
        ;                                     // walk queue
    *pp = b;                                // add request to end
    if (ide_queue == b)                      // if q is empty
        ide_start_request(b);               // send req to disk
    while ((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
        sleep(b, &ide_lock);                // wait for completion
    release(&ide_lock);
}

void ide_intr() {
    struct buf *b;
    acquire(&ide_lock);
    if (!(b->flags & B_DIRTY) && ide_wait_ready() >= 0)
        insl(0x1f0, b->data, 512/4);     // if READ: get data
    b->flags |= B_VALID;
    b->flags &= ~B_DIRTY;
    wakeup(b);                            // wake waiting process
    if ((ide_queue = b->qnext) != 0) // start next request
        ide_start_request(ide_queue); // (if one exists)
    release(&ide_lock);
}

```

**Figure 36.5: The xv6 IDE Disk Driver (Simplified)**

case, the routine waits for the request to complete and the calling process is put to sleep. The second is `ide_start_request()`, which is used to send a request (and perhaps data, in the case of a write) to the disk; the `in` and `out` x86 instructions are called to read and write device registers, respectively. The start request routine uses the third function, `ide_wait_ready()`, to ensure the drive is ready before issuing a request to it. Finally, `ide_intr()` is invoked when an interrupt takes place; it reads data from the device (if the request is a read, not a write), wakes the process waiting for the I/O to complete, and (if there are more requests in the I/O queue), launches the next I/O via `ide_start_request()`.

### 36.9 Historical Notes

Before ending, we include a brief historical note on the origin of some of these fundamental ideas. If you are interested in learning more, read Smotherman's excellent summary [S08].

Interrupts are an ancient idea, existing on the earliest of machines. For example, the UNIVAC in the early 1950's had some form of interrupt vectoring, although it is unclear in exactly which year this feature was available [S08]. Sadly, even in its infancy, we are beginning to lose the origins of computing history.

There is also some debate as to which machine first introduced the idea of DMA. For example, Knuth and others point to the DYSEAC (a "mobile" machine, which at the time meant it could be hauled in a trailer), whereas others think the IBM SAGE may have been the first [S08]. Either way, by the mid 50's, systems with I/O devices that communicated directly with memory and interrupted the CPU when finished existed.

The history here is difficult to trace because the inventions are tied to real, and sometimes obscure, machines. For example, some think that the Lincoln Labs TX-2 machine was first with vectored interrupts [S08], but this is hardly clear.

Because the ideas are relatively obvious — no Einsteinian leap is required to come up with the idea of letting the CPU do something else while a slow I/O is pending — perhaps our focus on "who first?" is misguided. What is certainly clear: as people built these early machines, it became obvious that I/O support was needed. Interrupts, DMA, and related ideas are all direct outcomes of the nature of fast CPUs and slow devices; if you were there at the time, you might have had similar ideas.

### 36.10 Summary

You should now have a very basic understanding of how an OS interacts with a device. Two techniques, the interrupt and DMA, have been introduced to help with device efficiency, and two approaches to accessing device registers, explicit I/O instructions and memory-mapped I/O, have been described. Finally, the notion of a device driver has been presented, showing how the OS itself can encapsulate low-level details and thus make it easier to build the rest of the OS in a device-neutral fashion.

## References

- [A+11] "vIC: Interrupt Coalescing for Virtual Machine Storage Device IO"  
 Irfan Ahmad, Ajay Gulati, Ali Mashtizadeh  
 USENIX '11  
*A terrific survey of interrupt coalescing in traditional and virtualized environments.*
- [C01] "An Empirical Study of Operating System Errors"  
 Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, Dawson Engler  
 SOSP '01  
*One of the first papers to systematically explore how many bugs are in modern operating systems. Among other neat findings, the authors show that device drivers have something like seven times more bugs than mainline kernel code.*
- [CK+08] "The xv6 Operating System"  
 Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich  
 From: <http://pdos.csail.mit.edu/6.828/2008/index.html>  
*See ide.c for the IDE device driver, with a few more details therein.*
- [D07] "What Every Programmer Should Know About Memory"  
 Ulrich Drepper  
 November, 2007  
 Available: <http://www.akkadia.org/drepper/cpumemory.pdf>  
*A fantastic read about modern memory systems, starting at DRAM and going all the way up to virtualization and cache-optimized algorithms.*
- [G08] "EIO: Error-handling is Occasionally Correct"  
 Haryadi Gunawi, Cindy Rubio-Gonzalez, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Ben Liblit  
 FAST '08, San Jose, CA, February 2008  
*Our own work on building a tool to find code in Linux file systems that does not handle error return properly. We found hundreds and hundreds of bugs, many of which have now been fixed.*
- [L94] "AT Attachment Interface for Disk Drives"  
 Lawrence J. Lamers, X3T10 Technical Editor  
 Available: <ftp://ftp.t10.org/t13/project/d0791r4c-ATA-1.pdf>  
*Reference number: ANSI X3.221 - 1994 A rather dry document about device interfaces. Read it at your own peril.*
- [MR96] "Eliminating Receive Livelock in an Interrupt-driven Kernel"  
 Jeffrey Mogul and K. K. Ramakrishnan  
 USENIX '96, San Diego, CA, January 1996  
*Mogul and colleagues did a great deal of pioneering work on web server network performance. This paper is but one example.*
- [S08] "Interrupts"  
 Mark Smotherman, as of July '08  
 Available: <http://people.cs.clemson.edu/~mark/interrupts.html>  
*A treasure trove of information on the history of interrupts, DMA, and related early ideas in computing.*

[S03] "Improving the Reliability of Commodity Operating Systems"

Michael M. Swift, Brian N. Bershad, and Henry M. Levy

SOSP '03

*Swift's work revived interest in a more microkernel-like approach to operating systems; minimally, it finally gave some good reasons why address-space based protection could be useful in a modern OS.*

[W10] "Hard Disk Driver"

Washington State Course Homepage

Available: <http://eecs.wsu.edu/~cs460/cs560/HDDriver.html>

*A nice summary of a simple IDE disk drive's interface and how to build a device driver for it.*

## Hard Disk Drives

The last chapter introduced the general concept of an I/O device and showed you how the OS might interact with such a beast. In this chapter, we dive into more detail about one device in particular: the **hard disk drive**. These drives have been the main form of persistent data storage in computer systems for decades and much of the development of file system technology (coming soon) is predicated on their behavior. Thus, it is worth understanding the details of a disk's operation before building the file system software that manages it. Many of these details are available in excellent papers by Ruemmler and Wilkes [RW92] and Anderson, Dykes, and Riedel [ADR03].

### CRUX: HOW TO STORE AND ACCESS DATA ON DISK

How do modern hard-disk drives store data? What is the interface? How is the data actually laid out and accessed? How does disk scheduling improve performance?

### 37.1 The Interface

Let's start by understanding the interface to a modern disk drive. The basic interface for all modern drives is straightforward. The drive consists of a large number of sectors (512-byte blocks), each of which can be read or written. The sectors are numbered from 0 to  $n - 1$  on a disk with  $n$  sectors. Thus, we can view the disk as an array of sectors; 0 to  $n - 1$  is thus the **address space** of the drive.

Multi-sector operations are possible; indeed, many file systems will read or write 4KB at a time (or more). However, when updating the disk, the only guarantee drive manufacturers make is that a single 512-byte write is **atomic** (i.e., it will either complete in its entirety or it won't complete at all); thus, if an untimely power loss occurs, only a portion of a larger write may complete (sometimes called a **torn write**).

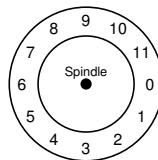


Figure 37.1: A Disk With Just A Single Track

There are some assumptions most clients of disk drives make, but that are not specified directly in the interface; Schlosser and Ganger have called this the “unwritten contract” of disk drives [SG04]. Specifically, one can usually assume that accessing two blocks that are near one-another within the drive’s address space will be faster than accessing two blocks that are far apart. One can also usually assume that accessing blocks in a contiguous chunk (i.e., a sequential read or write) is the fastest access mode, and usually much faster than any more random access pattern.

## 37.2 Basic Geometry

Let’s start to understand some of the components of a modern disk. We start with a **platter**, a circular hard surface on which data is stored persistently by inducing magnetic changes to it. A disk may have one or more platters; each platter has 2 sides, each of which is called a **surface**. These platters are usually made of some hard material (such as aluminum), and then coated with a thin magnetic layer that enables the drive to persistently store bits even when the drive is powered off.

The platters are all bound together around the **spindle**, which is connected to a motor that spins the platters around (while the drive is powered on) at a constant (fixed) rate. The rate of rotation is often measured in **rotations per minute (RPM)**, and typical modern values are in the 7,200 RPM to 15,000 RPM range. Note that we will often be interested in the time of a single rotation, e.g., a drive that rotates at 10,000 RPM means that a single rotation takes about 6 milliseconds (6 ms).

Data is encoded on each surface in concentric circles of sectors; we call one such concentric circle a **track**. A single surface contains many thousands and thousands of tracks, tightly packed together, with hundreds of tracks fitting into the width of a human hair.

To read and write from the surface, we need a mechanism that allows us to either sense (i.e., read) the magnetic patterns on the disk or to induce a change in (i.e., write) them. This process of reading and writing is accomplished by the **disk head**; there is one such head per surface of the drive. The disk head is attached to a single **disk arm**, which moves across the surface to position the head over the desired track.

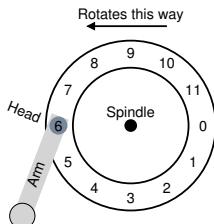


Figure 37.2: A Single Track Plus A Head

### 37.3 A Simple Disk Drive

Let's understand how disks work by building up a model one track at a time. Assume we have a simple disk with a single track (Figure 37.1).

This track has just 12 sectors, each of which is 512 bytes in size (our typical sector size, recall) and addressed therefore by the numbers 0 through 11. The single platter we have here rotates around the spindle, to which a motor is attached. Of course, the track by itself isn't too interesting; we want to be able to read or write those sectors, and thus we need a disk head, attached to a disk arm, as we now see (Figure 37.2).

In the figure, the disk head, attached to the end of the arm, is positioned over sector 6, and the surface is rotating counter-clockwise.

#### Single-track Latency: The Rotational Delay

To understand how a request would be processed on our simple, one-track disk, imagine we now receive a request to read block 0. How should the disk service this request?

In our simple disk, the disk doesn't have to do much. In particular, it must just wait for the desired sector to rotate under the disk head. This wait happens often enough in modern drives, and is an important enough component of I/O service time, that it has a special name: **rotational delay** (sometimes **rotation delay**, though that sounds weird). In the example, if the full rotational delay is  $R$ , the disk has to incur a rotational delay of about  $\frac{R}{2}$  to wait for 0 to come under the read/write head (if we start at 6). A worst-case request on this single track would be to sector 5, causing nearly a full rotational delay in order to service such a request.

#### Multiple Tracks: Seek Time

So far our disk just has a single track, which is not too realistic; modern disks of course have many millions. Let's thus look at ever-so-slightly more realistic disk surface, this one with three tracks (Figure 37.3, left).

In the figure, the head is currently positioned over the innermost track (which contains sectors 24 through 35); the next track over contains the next set of sectors (12 through 23), and the outermost track contains the first sectors (0 through 11).

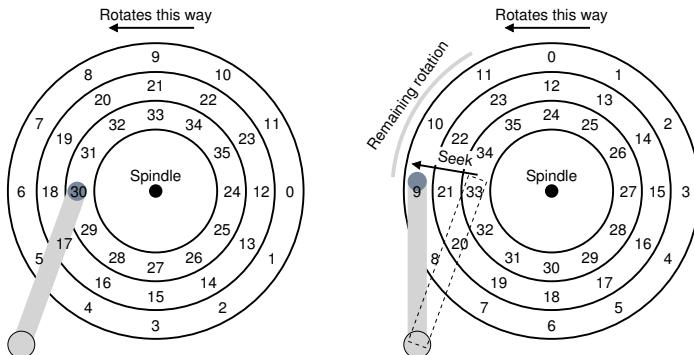


Figure 37.3: Three Tracks Plus A Head (Right: With Seek)

To understand how the drive might access a given sector, we now trace what would happen on a request to a distant sector, e.g., a read to sector 11. To service this read, the drive has to first move the disk arm to the correct track (in this case, the outermost one), in a process known as a **seek**. Seek, along with rotations, are one of the most costly disk operations.

The seek, it should be noted, has many phases: first an *acceleration* phase as the disk arm gets moving; then *coasting* as the arm is moving at full speed, then *deceleration* as the arm slows down; finally *settling* as the head is carefully positioned over the correct track. The **settling time** is often quite significant, e.g., 0.5 to 2 ms, as the drive must be certain to find the right track (imagine if it just got close instead!).

After the seek, the disk arm has positioned the head over the right track. A depiction of the seek is found in Figure 37.3 (right).

As we can see, during the seek, the arm has been moved to the desired track, and the platter of course has rotated, in this case about 3 sectors. Thus, sector 9 is just about to pass under the disk head, and we must only endure a short rotational delay to complete the transfer.

When sector 11 passes under the disk head, the final phase of I/O will take place, known as the **transfer**, where data is either read from or written to the surface. And thus, we have a complete picture of I/O time: first a seek, then waiting for the rotational delay, and finally the transfer.

## Some Other Details

Though we won't spend too much time on it, there are some other interesting details about how hard drives operate. Many drives employ some kind of **track skew** to make sure that sequential reads can be properly serviced even when crossing track boundaries. In our simple example disk, this might appear as seen in Figure 37.4.

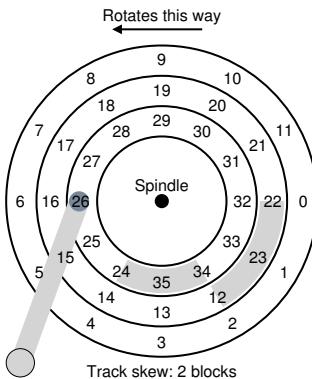


Figure 37.4: Three Tracks: Track Skew Of 2

Sectors are often skewed like this because when switching from one track to another, the disk needs time to reposition the head (even to neighboring tracks). Without such skew, the head would be moved to the next track but the desired next block would have already rotated under the head, and thus the drive would have to wait almost the entire rotational delay to access the next block.

Another reality is that outer tracks tend to have more sectors than inner tracks, which is a result of geometry; there is simply more room out there. These tracks are often referred to as **multi-zoned** disk drives, where the disk is organized into multiple zones, and where a zone is consecutive set of tracks on a surface. Each zone has the same number of sectors per track, and outer zones have more sectors than inner zones.

Finally, an important part of any modern disk drive is its **cache**, for historical reasons sometimes called a **track buffer**. This cache is just some small amount of memory (usually around 8 or 16 MB) which the drive can use to hold data read from or written to the disk. For example, when reading a sector from the disk, the drive might decide to read in all of the sectors on that track and cache them in its memory; doing so allows the drive to quickly respond to any subsequent requests to the same track.

On writes, the drive has a choice: should it acknowledge the write has completed when it has put the data in its memory, or after the write has actually been written to disk? The former is called **write back** caching (or sometimes **immediate reporting**), and the latter **write through**. Write back caching sometimes makes the drive appear “faster”, but can be dangerous; if the file system or applications require that data be written to disk in a certain order for correctness, write-back caching can lead to problems (read the chapter on file-system journaling for details).

### ASIDE: DIMENSIONAL ANALYSIS

Remember in Chemistry class, how you solved virtually every problem by simply setting up the units such that they canceled out, and somehow the answers popped out as a result? That chemical magic is known by the highfalutin name of **dimensional analysis** and it turns out it is useful in computer systems analysis too.

Let's do an example to see how dimensional analysis works and why it is useful. In this case, assume you have to figure out how long, in milliseconds, a single rotation of a disk takes. Unfortunately, you are given only the **RPM** of the disk, or **rotations per minute**. Let's assume we're talking about a 10K RPM disk (i.e., it rotates 10,000 times per minute). How do we set up the dimensional analysis so that we get time per rotation in milliseconds?

To do so, we start by putting the desired units on the left; in this case, we wish to obtain the time (in milliseconds) per rotation, so that is exactly what we write down:  $\frac{\text{Time (ms)}}{1 \text{ Rotation}}$ . We then write down everything we know, making sure to cancel units where possible. First, we obtain  $\frac{1 \text{ minute}}{10,000 \text{ Rotations}}$  (keeping rotation on the bottom, as that's where it is on the left), then transform minutes into seconds with  $\frac{60 \text{ seconds}}{1 \text{ minute}}$ , and then finally transform seconds in milliseconds with  $\frac{1000 \text{ ms}}{1 \text{ second}}$ . The final result is the following (with units nicely canceled):

$$\frac{\text{Time (ms)}}{1 \text{ Rot.}} = \frac{1 \text{ minute}}{10,000 \text{ Rot.}} \cdot \frac{60 \text{ seconds}}{1 \text{ minute}} \cdot \frac{1000 \text{ ms}}{1 \text{ second}} = \frac{60,000 \text{ ms}}{10,000 \text{ Rot.}} = \frac{6 \text{ ms}}{\text{Rotation}}$$

As you can see from this example, dimensional analysis makes what seems intuitive into a simple and repeatable process. Beyond the RPM calculation above, it comes in handy with I/O analysis regularly. For example, you will often be given the transfer rate of a disk, e.g., 100 MB/second, and then asked: how long does it take to transfer a 512 KB block (in milliseconds)? With dimensional analysis, it's easy:

$$\frac{\text{Time (ms)}}{1 \text{ Request}} = \frac{512 \text{ KB}}{1 \text{ Request}} \cdot \frac{1 \text{ MB}}{1024 \text{ KB}} \cdot \frac{1 \text{ second}}{100 \text{ MB}} \cdot \frac{1000 \text{ ms}}{1 \text{ second}} = \frac{5 \text{ ms}}{\text{Request}}$$

## 37.4 I/O Time: Doing The Math

Now that we have an abstract model of the disk, we can use a little analysis to better understand disk performance. In particular, we can now represent I/O time as the sum of three major components:

$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer} \tag{37.1}$$

	Cheetah 15K.5	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Average Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	16/32 MB
Connects via	SCSI	SATA

Figure 37.5: Disk Drive Specs: SCSI Versus SATA

Note that the rate of I/O ( $R_{I/O}$ ), which is often more easily used for comparison between drives (as we will do below), is easily computed from the time. Simply divide the size of the transfer by the time it took:

$$R_{I/O} = \frac{\text{Size}_{\text{Transfer}}}{T_{I/O}} \quad (37.2)$$

To get a better feel for I/O time, let us perform the following calculation. Assume there are two workloads we are interested in. The first, known as the **random** workload, issues small (e.g., 4KB) reads to random locations on the disk. Random workloads are common in many important applications, including database management systems. The second, known as the **sequential** workload, simply reads a large number of sectors consecutively from the disk, without jumping around. Sequential access patterns are quite common and thus important as well.

To understand the difference in performance between random and sequential workloads, we need to make a few assumptions about the disk drive first. Let's look at a couple of modern disks from Seagate. The first, known as the Cheetah 15K.5 [S09b], is a high-performance SCSI drive. The second, the Barracuda [S09a], is a drive built for capacity. Details on both are found in Figure 37.5.

As you can see, the drives have quite different characteristics, and in many ways nicely summarize two important components of the disk drive market. The first is the “high performance” drive market, where drives are engineered to spin as fast as possible, deliver low seek times, and transfer data quickly. The second is the “capacity” market, where cost per byte is the most important aspect; thus, the drives are slower but pack as many bits as possible into the space available.

From these numbers, we can start to calculate how well the drives would do under our two workloads outlined above. Let's start by looking at the random workload. Assuming each 4 KB read occurs at a random location on disk, we can calculate how long each such read would take. On the Cheetah:

$$T_{\text{seek}} = 4 \text{ ms}, T_{\text{rotation}} = 2 \text{ ms}, T_{\text{transfer}} = 30 \text{ microsecs} \quad (37.3)$$

**TIP: USE DISKS SEQUENTIALLY**

When at all possible, transfer data to and from disks in a sequential manner. If sequential is not possible, at least think about transferring data in large chunks: the bigger, the better. If I/O is done in little random pieces, I/O performance will suffer dramatically. Also, users will suffer. Also, you will suffer, knowing what suffering you have wrought with your careless random I/Os.

The average seek time (4 milliseconds) is just taken as the average time reported by the manufacturer; note that a full seek (from one end of the surface to the other) would likely take two or three times longer. The average rotational delay is calculated from the RPM directly. 15000 RPM is equal to 250 RPS (rotations per second); thus, each rotation takes 4 ms. On average, the disk will encounter a half rotation and thus 2 ms is the average time. Finally, the transfer time is just the size of the transfer over the peak transfer rate; here it is vanishingly small (30 microseconds; note that we need 1000 microseconds just to get 1 millisecond!).

Thus, from our equation above,  $T_{I/O}$  for the Cheetah roughly equals 6 ms. To compute the rate of I/O, we just divide the size of the transfer by the average time, and thus arrive at  $R_{I/O}$  for the Cheetah under the random workload of about 0.66 MB/s. The same calculation for the Barracuda yields a  $T_{I/O}$  of about 13.2 ms, more than twice as slow, and thus a rate of about 0.31 MB/s.

Now let's look at the sequential workload. Here we can assume there is a single seek and rotation before a very long transfer. For simplicity, assume the size of the transfer is 100 MB. Thus,  $T_{I/O}$  for the Barracuda and Cheetah is about 800 ms and 950 ms, respectively. The rates of I/O are thus very nearly the peak transfer rates of 125 MB/s and 105 MB/s, respectively. Figure 37.6 summarizes these numbers.

The figure shows us a number of important things. First, and most importantly, there is a huge gap in drive performance between random and sequential workloads, almost a factor of 200 or so for the Cheetah and more than a factor 300 difference for the Barracuda. And thus we arrive at the most obvious design tip in the history of computing.

A second, more subtle point: there is a large difference in performance between high-end “performance” drives and low-end “capacity” drives. For this reason (and others), people are often willing to pay top dollar for the former while trying to get the latter as cheaply as possible.

	Cheetah	Barracuda
$R_{I/O}$ Random	0.66 MB/s	0.31 MB/s
$R_{I/O}$ Sequential	125 MB/s	105 MB/s

Figure 37.6: Disk Drive Performance: SCSI Versus SATA

**ASIDE: COMPUTING THE “AVERAGE” SEEK**

In many books and papers, you will see average disk-seek time cited as being roughly one-third of the full seek time. Where does this come from?

Turns out it arises from a simple calculation based on average seek *distance*, not time. Imagine the disk as a set of tracks, from 0 to  $N$ . The seek distance between any two tracks  $x$  and  $y$  is thus computed as the absolute value of the difference between them:  $|x - y|$ .

To compute the average seek distance, all you need to do is to first add up all possible seek distances:

$$\sum_{x=0}^N \sum_{y=0}^N |x - y|. \quad (37.4)$$

Then, divide this by the number of different possible seeks:  $N^2$ . To compute the sum, we'll just use the integral form:

$$\int_{x=0}^N \int_{y=0}^N |x - y| dy dx. \quad (37.5)$$

To compute the inner integral, let's break out the absolute value:

$$\int_{y=0}^x (x - y) dy + \int_{y=x}^N (y - x) dy. \quad (37.6)$$

Solving this leads to  $(xy - \frac{1}{2}y^2)|_0^x + (\frac{1}{2}y^2 - xy)|_x^N$  which can be simplified to  $(x^2 - Nx + \frac{1}{2}N^2)$ . Now we have to compute the outer integral:

$$\int_{x=0}^N (x^2 - Nx + \frac{1}{2}N^2) dx, \quad (37.7)$$

which results in:

$$\left( \frac{1}{3}x^3 - \frac{N}{2}x^2 + \frac{N^2}{2}x \right) \Big|_0^N = \frac{N^3}{3}. \quad (37.8)$$

Remember that we still have to divide by the total number of seeks ( $N^2$ ) to compute the average seek distance:  $(\frac{N^3}{3})/(N^2) = \frac{1}{3}N$ . Thus the average seek distance on a disk, over all possible seeks, is one-third the full distance. And now when you hear that an average seek is one-third of a full seek, you'll know where it came from.

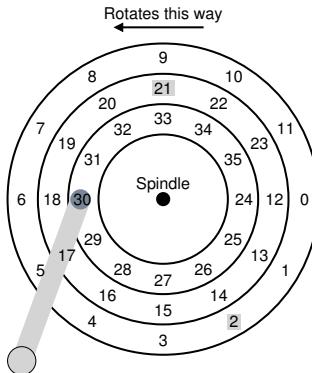


Figure 37.7: SSTF: Scheduling Requests 21 And 2

## 37.5 Disk Scheduling

Because of the high cost of I/O, the OS has historically played a role in deciding the order of I/Os issued to the disk. More specifically, given a set of I/O requests, the **disk scheduler** examines the requests and decides which one to schedule next [SCO90, JW91].

Unlike job scheduling, where the length of each job is usually unknown, with disk scheduling, we can make a good guess at how long a “job” (i.e., disk request) will take. By estimating the seek and possible rotational delay of a request, the disk scheduler can know how long each request will take, and thus (greedily) pick the one that will take the least time to service first. Thus, the disk scheduler will try to follow the **principle of SJF (shortest job first)** in its operation.

### SSTF: Shortest Seek Time First

One early disk scheduling approach is known as **shortest-seek-time-first (SSTF)** (also called **shortest-seek-first** or **SSF**). SSTF orders the queue of I/O requests by track, picking requests on the nearest track to complete first. For example, assuming the current position of the head is over the inner track, and we have requests for sectors 21 (middle track) and 2 (outer track), we would then issue the request to 21 first, wait for it to complete, and then issue the request to 2 (Figure 37.7).

SSTF works well in this example, seeking to the middle track first and then the outer track. However, SSTF is not a panacea, for the following reasons. First, the drive geometry is not available to the host OS; rather, it sees an array of blocks. Fortunately, this problem is rather easily fixed. Instead of SSTF, an OS can simply implement **nearest-block-first (NBF)**, which schedules the request with the nearest block address next.

The second problem is more fundamental: **starvation**. Imagine in our example above if there were a steady stream of requests to the inner track, where the head currently is positioned. Requests to any other tracks would then be ignored completely by a pure SSTF approach. And thus the crux of the problem:

#### CRUX: HOW TO HANDLE DISK STARVATION

How can we implement SSTF-like scheduling but avoid starvation?

### Elevator (a.k.a. SCAN or C-SCAN)

The answer to this query was developed some time ago (see [CKR72] for example), and is relatively straightforward. The algorithm, originally called **SCAN**, simply moves back and forth across the disk servicing requests in order across the tracks. Let's call a single pass across the disk (from outer to inner tracks, or inner to outer) a *sweep*. Thus, if a request comes for a block on a track that has already been serviced on this sweep of the disk, it is not handled immediately, but rather queued until the next sweep (in the other direction).

SCAN has a number of variants, all of which do about the same thing. For example, Coffman et al. introduced **F-SCAN**, which freezes the queue to be serviced when it is doing a sweep [CKR72]; this action places requests that come in during the sweep into a queue to be serviced later. Doing so avoids starvation of far-away requests, by delaying the servicing of late-arriving (but nearer by) requests.

**C-SCAN** is another common variant, short for **Circular SCAN**. Instead of sweeping in both directions across the disk, the algorithm only sweeps from outer-to-inner, and then resets at the outer track to begin again. Doing so is a bit more fair to inner and outer tracks, as pure back-and-forth SCAN favors the middle tracks, i.e., after servicing the outer track, SCAN passes through the middle twice before coming back to the outer track again.

For reasons that should now be clear, the SCAN algorithm (and its cousins) is sometimes referred to as the **elevator** algorithm, because it behaves like an elevator which is either going up or down and not just servicing requests to floors based on which floor is closer. Imagine how annoying it would be if you were going down from floor 10 to 1, and somebody got on at 3 and pressed 4, and the elevator went up to 4 because it was "closer" than 1! As you can see, the elevator algorithm, when used in real life, prevents fights from taking place on elevators. In disks, it just prevents starvation.

Unfortunately, SCAN and its cousins do not represent the best scheduling technology. In particular, SCAN (or SSTF even) do not actually adhere as closely to the principle of SJF as they could. In particular, they ignore rotation. And thus, another crux:

### CRUX: HOW TO ACCOUNT FOR DISK ROTATION COSTS

How can we implement an algorithm that more closely approximates SJF by taking *both* seek and rotation into account?

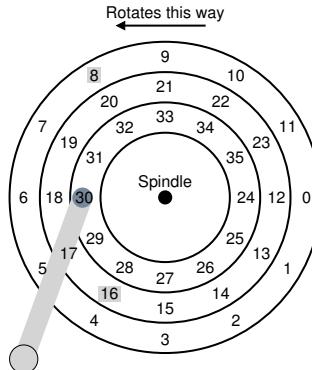


Figure 37.8: SSTF: Sometimes Not Good Enough  
SPTF: Shortest Positioning Time First

Before discussing **shortest positioning time first** or **SPTF** scheduling (sometimes also called **shortest access time first** or **SATF**), which is the solution to our problem, let us make sure we understand the problem in more detail. Figure 37.8 presents an example.

In the example, the head is currently positioned over sector 30 on the inner track. The scheduler thus has to decide: should it schedule sector 16 (on the middle track) or sector 8 (on the outer track) for its next request. So which should it service next?

The answer, of course, is “it depends”. In engineering, it turns out “it depends” is almost always the answer, reflecting that trade-offs are part of the life of the engineer; such maxims are also good in a pinch, e.g., when you don’t know an answer to your boss’s question, you might want to try this gem. However, it is almost always better to know *why* it depends, which is what we discuss here.

What it depends on here is the relative time of seeking as compared to rotation. If, in our example, seek time is much higher than rotational delay, then SSTF (and variants) are just fine. However, imagine if seek is quite a bit faster than rotation. Then, in our example, it would make more sense to seek *further* to service request 8 on the outer track than it would to perform the shorter seek to the middle track to service 16, which has to rotate all the way around before passing under the disk head.

On modern drives, as we saw above, both seek and rotation are roughly

**TIP: IT ALWAYS DEPENDS (LIVNY'S LAW)**

Almost any question can be answered with “it depends”, as our colleague Miron Livny always says. However, use with caution, as if you answer too many questions this way, people will stop asking you questions altogether. For example, somebody asks: “want to go to lunch?” You reply: “it depends, are *you* coming along?”

equivalent (depending, of course, on the exact requests), and thus SPTF is useful and improves performance. However, it is even more difficult to implement in an OS, which generally does not have a good idea where track boundaries are or where the disk head currently is (in a rotational sense). Thus, SPTF is usually performed inside a drive, described below.

## Other Scheduling Issues

There are many other issues we do not discuss in this brief description of basic disk operation, scheduling, and related topics. One such issue is this: *where* is disk scheduling performed on modern systems? In older systems, the operating system did all the scheduling; after looking through the set of pending requests, the OS would pick the best one, and issue it to the disk. When that request completed, the next one would be chosen, and so forth. Disks were simpler then, and so was life.

In modern systems, disks can accommodate multiple outstanding requests, and have sophisticated internal schedulers themselves (which can implement SPTF accurately; inside the disk controller, all relevant details are available, including exact head position). Thus, the OS scheduler usually picks what it thinks the best few requests are (say 16) and issues them all to disk; the disk then uses its internal knowledge of head position and detailed track layout information to service said requests in the best possible (SPTF) order.

Another important related task performed by disk schedulers is **I/O merging**. For example, imagine a series of requests to read blocks 33, then 8, then 34, as in Figure 37.8. In this case, the scheduler should **merge** the requests for blocks 33 and 34 into a single two-block request; any re-ordering that the scheduler does is performed upon the merged requests. Merging is particularly important at the OS level, as it reduces the number of requests sent to the disk and thus lowers overheads.

One final problem that modern schedulers address is this: how long should the system wait before issuing an I/O to disk? One might naively think that the disk, once it has even a single I/O, should immediately issue the request to the drive; this approach is called **work-conserving**, as the disk will never be idle if there are requests to serve. However, research on **anticipatory disk scheduling** has shown that sometimes it is better to wait for a bit [ID01], in what is called a **non-work-conserving** approach.

By waiting, a new and “better” request may arrive at the disk, and thus overall efficiency is increased. Of course, deciding when to wait, and for how long, can be tricky; see the research paper for details, or check out the Linux kernel implementation to see how such ideas are transitioned into practice (if you are the ambitious sort).

## 37.6 Summary

We have presented a summary of how disks work. The summary is actually a detailed functional model; it does not describe the amazing physics, electronics, and material science that goes into actual drive design. For those interested in even more details of that nature, we suggest a different major (or perhaps minor); for those that are happy with this model, good! We can now proceed to using the model to build more interesting systems on top of these incredible devices.

## References

- [ADR03] "More Than an Interface: SCSI vs. ATA"  
 Dave Anderson, Jim Dykes, Erik Riedel  
 FAST '03, 2003  
*One of the best recent-ish references on how modern disk drives really work; a must read for anyone interested in knowing more.*
- [CKR72] "Analysis of Scanning Policies for Reducing Disk Seek Times"  
 E.G. Coffman, L.A. Klimko, B. Ryan  
 SIAM Journal of Computing, September 1972, Vol 1. No 3.  
*Some of the early work in the field of disk scheduling.*
- [ID01] "Anticipatory Scheduling: A Disk-scheduling Framework To Overcome Deceptive Idleness In Synchronous I/O"  
 Sitaram Iyer, Peter Druschel  
 SOSP '01, October 2001  
*A cool paper showing how waiting can improve disk scheduling: better requests may be on their way!*
- [JW91] "Disk Scheduling Algorithms Based On Rotational Position"  
 D. Jacobson, J. Wilkes  
 Technical Report HPL-CSP-91-7rev1, Hewlett-Packard (February 1991)  
*A more modern take on disk scheduling. It remains a technical report (and not a published paper) because the authors were scooped by Seltzer et al. [SCO90].*
- [RW92] "An Introduction to Disk Drive Modeling"  
 C. Ruemmler, J. Wilkes  
 IEEE Computer, 27:3, pp. 17-28, March 1994  
*A terrific introduction to the basics of disk operation. Some pieces are out of date, but most of the basics remain.*
- [SCO90] "Disk Scheduling Revisited"  
 Margo Seltzer, Peter Chen, John Ousterhout  
 USENIX 1990  
*A paper that talks about how rotation matters too in the world of disk scheduling.*
- [SG04] "MEMS-based storage devices and standard disk interfaces:  
 A square peg in a round hole?"  
 Steven W. Schlosser, Gregory R. Ganger  
 FAST '04, pp. 87-100, 2004  
*While the MEMS aspect of this paper hasn't yet made an impact, the discussion of the contract between file systems and disks is wonderful and a lasting contribution.*
- [S09a] "Barracuda ES.2 data sheet"  
[http://www.seagate.com/docs/pdf/datasheet/disc/ds\\_cheetah\\_15k\\_5.pdf](http://www.seagate.com/docs/pdf/datasheet/disc/ds_cheetah_15k_5.pdf) *A data sheet; read at your own risk. Risk of what? Boredom.*
- [S09b] "Cheetah 15K.5"  
[http://www.seagate.com/docs/pdf/datasheet/disc/ds\\_barracuda\\_es.pdf](http://www.seagate.com/docs/pdf/datasheet/disc/ds_barracuda_es.pdf) *See above commentary on data sheets.*

## Homework

This homework uses `disk.py` to familiarize you with how a modern hard drive works. It has a lot of different options, and unlike most of the other simulations, has a graphical animator to show you exactly what happens when the disk is in action. See the README for details.

1. Compute the seek, rotation, and transfer times for the following sets of requests: `-a 0, -a 6, -a 30, -a 7, 30, 8`, and finally `-a 10, 11, 12, 13`.
2. Do the same requests above, but change the seek rate to different values: `-S 2, -S 4, -S 8, -S 10, -S 40, -S 0.1`. How do the times change?
3. Do the same requests above, but change the rotation rate: `-R 0.1, -R 0.5, -R 0.01`. How do the times change?
4. You might have noticed that some request streams would be better served with a policy better than FIFO. For example, with the request stream `-a 7, 30, 8`, what order should the requests be processed in? Now run the shortest seek-time first (SSTF) scheduler (`-p SSTF`) on the same workload; how long should it take (seek, rotation, transfer) for each request to be served?
5. Now do the same thing, but using the shortest access-time first (SATF) scheduler (`-p SATF`). Does it make any difference for the set of requests as specified by `-a 7, 30, 8`? Find a set of requests where SATF does noticeably better than SSTF; what are the conditions for a noticeable difference to arise?
6. You might have noticed that the request stream `-a 10, 11, 12, 13` wasn't particularly well handled by the disk. Why is that? Can you introduce a track skew to address this problem (`-o skew`, where `skew` is a non-negative integer)? Given the default seek rate, what should the skew be to minimize the total time for this set of requests? What about for different seek rates (e.g., `-S 2, -S 4`)? In general, could you write a formula to figure out the skew, given the seek rate and sector layout information?
7. Multi-zone disks pack more sectors into the outer tracks. To configure this disk in such a way, run with the `-z` flag. Specifically, try running some requests against a disk run with `-z 10, 20, 30` (the numbers specify the angular space occupied by a sector, per track; in this example, the outer track will be packed with a sector every 10 degrees, the middle track every 20 degrees, and the inner track with a sector every 30 degrees). Run some random requests (e.g., `-a -1 -A 5, -1, 0`, which specifies that random requests should be used via the `-a -1` flag and that five requests ranging from 0 to the max be generated), and see if you can compute the seek, rotation, and transfer times. Use different random seeds (`-s 1, -s 2`, etc.). What is the bandwidth (in sectors per unit time) on the outer, middle, and inner tracks?

8. Scheduling windows determine how many sector requests a disk can examine at once in order to determine which sector to serve next. Generate some random workloads of a lot of requests (e.g., `-A 1000, -1, 0`, with different seeds perhaps) and see how long the SATF scheduler takes when the scheduling window is changed from 1 up to the number of requests (e.g., `-w 1` up to `-w 1000`, and some values in between). How big of scheduling window is needed to approach the best possible performance? Make a graph and see. Hint: use the `-c` flag and don't turn on graphics with `-G` to run these more quickly. When the scheduling window is set to 1, does it matter which policy you are using?
9. Avoiding starvation is important in a scheduler. Can you think of a series of requests such that a particular sector is delayed for a very long time given a policy such as SATF? Given that sequence, how does it perform if you use a **bounded SATF** or **BSATF** scheduling approach? In this approach, you specify the scheduling window (e.g., `-w 4`) as well as the BSATF policy (`-p BSATF`); the scheduler then will only move onto the next window of requests when *all* of the requests in the current window have been serviced. Does this solve the starvation problem? How does it perform, as compared to SATF? In general, how should a disk make this trade-off between performance and starvation avoidance?
10. All the scheduling policies we have looked at thus far are **greedy**, in that they simply pick the next best option instead of looking for the optimal schedule over a set of requests. Can you find a set of requests in which this greedy approach is not optimal?

## Redundant Arrays of Inexpensive Disks (RAIDs)

When we use a disk, we sometimes wish it to be faster; I/O operations are slow and thus can be the bottleneck for the entire system. When we use a disk, we sometimes wish it to be larger; more and more data is being put online and thus our disks are getting fuller and fuller. When we use a disk, we sometimes wish for it to be more reliable; when a disk fails, if our data isn't backed up, all that valuable data is gone.

### CRUX: HOW TO MAKE A LARGE, FAST, RELIABLE DISK

How can we make a large, fast, and reliable storage system? What are the key techniques? What are trade-offs between different approaches?

In this chapter, we introduce the **Redundant Array of Inexpensive Disks** better known as **RAID** [P+88], a technique to use multiple disks in concert to build a faster, bigger, and more reliable disk system. The term was introduced in the late 1980s by a group of researchers at U.C. Berkeley (led by Professors David Patterson and Randy Katz and then student Garth Gibson); it was around this time that many different researchers simultaneously arrived upon the basic idea of using multiple disks to build a better storage system [BG88, K86, K88, PB86, SG86].

Externally, a RAID looks like a disk: a group of blocks one can read or write. Internally, the RAID is a complex beast, consisting of multiple disks, memory (both volatile and non-), and one or more processors to manage the system. A hardware RAID is very much like a computer system, specialized for the task of managing a group of disks.

RAIDS offer a number of advantages over a single disk. One advantage is *performance*. Using multiple disks in parallel can greatly speed up I/O times. Another benefit is *capacity*. Large data sets demand large disks. Finally, RAIDs can improve *reliability*; spreading data across multiple disks (without RAID techniques) makes the data vulnerable to the loss of a single disk; with some form of **redundancy**, RAIDs can tolerate the loss of a disk and keep operating as if nothing were wrong.

**TIP: TRANSPARENCY ENABLES DEPLOYMENT**

When considering how to add new functionality to a system, one should always consider whether such functionality can be added **transparently**, in a way that demands no changes to the rest of the system. Requiring a complete rewrite of the existing software (or radical hardware changes) lessens the chance of impact of an idea. RAID is a perfect example, and certainly its transparency contributed to its success; administrators could install a SCSI-based RAID storage array instead of a SCSI disk, and the rest of the system (host computer, OS, etc.) did not have to change one bit to start using it. By solving this problem of **deployment**, RAID was made more successful from day one.

Amazingly, RAIDs provide these advantages **transparently** to systems that use them, i.e., a RAID just looks like a big disk to the host system. The beauty of transparency, of course, is that it enables one to simply replace a disk with a RAID and not change a single line of software; the operating system and client applications continue to operate without modification. In this manner, transparency greatly improves the **deployability** of RAID, enabling users and administrators to put a RAID to use without worries of software compatibility.

We now discuss some of the important aspects of RAIDs. We begin with the interface, fault model, and then discuss how one can evaluate a RAID design along three important axes: capacity, reliability, and performance. We then discuss a number of other issues that are important to RAID design and implementation.

### 38.1 Interface And RAID Internals

To a file system above, a RAID looks like a big, (hopefully) fast, and (hopefully) reliable disk. Just as with a single disk, it presents itself as a linear array of blocks, each of which can be read or written by the file system (or other client).

When a file system issues a *logical I/O* request to the RAID, the RAID internally must calculate which disk (or disks) to access in order to complete the request, and then issue one or more *physical I/Os* to do so. The exact nature of these physical I/Os depends on the RAID level, as we will discuss in detail below. However, as a simple example, consider a RAID that keeps two copies of each block (each one on a separate disk); when writing to such a **mirrored** RAID system, the RAID will have to perform two physical I/Os for every one logical I/O it is issued.

A RAID system is often built as a separate hardware box, with a standard connection (e.g., SCSI, or SATA) to a host. Internally, however, RAIDs are fairly complex, consisting of a microcontroller that runs firmware to direct the operation of the RAID, volatile memory such as DRAM to buffer data blocks as they are read and written, and in some cases,

non-volatile memory to buffer writes safely and perhaps even specialized logic to perform parity calculations (useful in some RAID levels, as we will also see below). At a high level, a RAID is very much a specialized computer system: it has a processor, memory, and disks; however, instead of running applications, it runs specialized software designed to operate the RAID.

## 38.2 Fault Model

To understand RAID and compare different approaches, we must have a fault model in mind. RAIDs are designed to detect and recover from certain kinds of disk faults; thus, knowing exactly which faults to expect is critical in arriving upon a working design.

The first fault model we will assume is quite simple, and has been called the **fail-stop** fault model [S84]. In this model, a disk can be in exactly one of two states: working or failed. With a working disk, all blocks can be read or written. In contrast, when a disk has failed, we assume it is permanently lost.

One critical aspect of the fail-stop model is what it assumes about fault detection. Specifically, when a disk has failed, we assume that this is easily detected. For example, in a RAID array, we would assume that the RAID controller hardware (or software) can immediately observe when a disk has failed.

Thus, for now, we do not have to worry about more complex “silent” failures such as disk corruption. We also do not have to worry about a single block becoming inaccessible upon an otherwise working disk (sometimes called a latent sector error). We will consider these more complex (and unfortunately, more realistic) disk faults later.

## 38.3 How To Evaluate A RAID

As we will soon see, there are a number of different approaches to building a RAID. Each of these approaches has different characteristics which are worth evaluating, in order to understand their strengths and weaknesses.

Specifically, we will evaluate each RAID design along three axes. The first axis is **capacity**; given a set of  $N$  disks each with  $B$  blocks, how much useful capacity is available to clients of the RAID? Without redundancy, the answer is  $N \cdot B$ ; in contrast, if we have a system that keeps two copies of each block (called **mirroring**), we obtain a useful capacity of  $(N \cdot B)/2$ . Different schemes (e.g., parity-based ones) tend to fall in between.

The second axis of evaluation is **reliability**. How many disk faults can the given design tolerate? In alignment with our fault model, we assume only that an entire disk can fail; in later chapters (i.e., on data integrity), we’ll think about how to handle more complex failure modes.

Finally, the third axis is **performance**. Performance is somewhat chal-

lenging to evaluate, because it depends heavily on the workload presented to the disk array. Thus, before evaluating performance, we will first present a set of typical workloads that one should consider.

We now consider three important RAID designs: RAID Level 0 (striping), RAID Level 1 (mirroring), and RAID Levels 4/5 (parity-based redundancy). The naming of each of these designs as a “level” stems from the pioneering work of Patterson, Gibson, and Katz at Berkeley [P+88].

### 38.4 RAID Level 0: Striping

The first RAID level is actually not a RAID level at all, in that there is no redundancy. However, RAID level 0, or **striping** as it is better known, serves as an excellent upper-bound on performance and capacity and thus is worth understanding.

The simplest form of striping will **stripe** blocks across the disks of the system as follows (assume here a 4-disk array):

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 38.1: RAID-0: Simple Striping

From Figure 38.1, you get the basic idea: spread the blocks of the array across the disks in a round-robin fashion. This approach is designed to extract the most parallelism from the array when requests are made for contiguous chunks of the array (as in a large, sequential read, for example). We call the blocks in the same row a **stripe**; thus, blocks 0, 1, 2, and 3 are in the same stripe above.

In the example, we have made the simplifying assumption that only 1 block (each of say size 4KB) is placed on each disk before moving on to the next. However, this arrangement need not be the case. For example, we could arrange the blocks across disks as in Figure 38.2:

Disk 0	Disk 1	Disk 2	Disk 3	
0	2	4	6	chunk size:
1	3	5	7	2 blocks
8	10	12	14	
9	11	13	15	

Figure 38.2: Striping with a Bigger Chunk Size

In this example, we place two 4KB blocks on each disk before moving on to the next disk. Thus, the **chunk size** of this RAID array is 8KB, and a stripe thus consists of 4 chunks or 32KB of data.

**ASIDE: THE RAID MAPPING PROBLEM**

Before studying the capacity, reliability, and performance characteristics of the RAID, we first present an aside on what we call **the mapping problem**. This problem arises in all RAID arrays; simply put, given a logical block to read or write, how does the RAID know exactly which physical disk and offset to access?

For these simple RAID levels, we do not need much sophistication in order to correctly map logical blocks onto their physical locations. Take the first striping example above (chunk size = 1 block = 4KB). In this case, given a logical block address A, the RAID can easily compute the desired disk and offset with two simple equations:

$$\begin{aligned} \text{Disk} &= A \% \text{ number\_of\_disks} \\ \text{Offset} &= A / \text{number\_of\_disks} \end{aligned}$$

Note that these are all integer operations (e.g.,  $4 / 3 = 1$  not  $1.33333\dots$ ).

Let's see how these equations work for a simple example. Imagine in the first RAID above that a request arrives for block 14. Given that there are 4 disks, this would mean that the disk we are interested in is  $(14 \% 4 = 2)$ : disk 2. The exact block is calculated as  $(14 / 4 = 3)$ : block 3. Thus, block 14 should be found on the fourth block (block 3, starting at 0) of the third disk (disk 2, starting at 0), which is exactly where it is.

You can think about how these equations would be modified to support different chunk sizes. Try it! It's not too hard.

## Chunk Sizes

Chunk size mostly affects performance of the array. For example, a small chunk size implies that many files will get striped across many disks, thus increasing the parallelism of reads and writes to a single file; however, the positioning time to access blocks across multiple disks increases, because the positioning time for the entire request is determined by the maximum of the positioning times of the requests across all drives.

A big chunk size, on the other hand, reduces such intra-file parallelism, and thus relies on multiple concurrent requests to achieve high throughput. However, large chunk sizes reduce positioning time; if, for example, a single file fits within a chunk and thus is placed on a single disk, the positioning time incurred while accessing it will just be the positioning time of a single disk.

Thus, determining the "best" chunk size is hard to do, as it requires a great deal of knowledge about the workload presented to the disk system [CL95]. For the rest of this discussion, we will assume that the array uses a chunk size of a single block (4KB). Most arrays use larger chunk sizes (e.g., 64 KB), but for the issues we discuss below, the exact chunk size does not matter; thus we use a single block for the sake of simplicity.

## Back To RAID-0 Analysis

Let us now evaluate the capacity, reliability, and performance of striping. From the perspective of capacity, it is perfect: given  $N$  disks each of size  $B$  blocks, striping delivers  $N \cdot B$  blocks of useful capacity. From the standpoint of reliability, striping is also perfect, but in the bad way: any disk failure will lead to data loss. Finally, performance is excellent: all disks are utilized, often in parallel, to service user I/O requests.

## Evaluating RAID Performance

In analyzing RAID performance, one can consider two different performance metrics. The first is *single-request latency*. Understanding the latency of a single I/O request to a RAID is useful as it reveals how much parallelism can exist during a single logical I/O operation. The second is *steady-state throughput* of the RAID, i.e., the total bandwidth of many concurrent requests. Because RAIDs are often used in high-performance environments, the steady-state bandwidth is critical, and thus will be the main focus of our analyses.

To understand throughput in more detail, we need to put forth some workloads of interest. We will assume, for this discussion, that there are two types of workloads: **sequential** and **random**. With a sequential workload, we assume that requests to the array come in large contiguous chunks; for example, a request (or series of requests) that accesses 1 MB of data, starting at block  $x$  and ending at block  $(x+1 \text{ MB})$ , would be deemed sequential. Sequential workloads are common in many environments (think of searching through a large file for a keyword), and thus are considered important.

For random workloads, we assume that each request is rather small, and that each request is to a different random location on disk. For example, a random stream of requests may first access 4KB at logical address 10, then at logical address 550,000, then at 20,100, and so forth. Some important workloads, such as transactional workloads on a database management system (DBMS), exhibit this type of access pattern, and thus it is considered an important workload.

Of course, real workloads are not so simple, and often have a mix of sequential and random-seeming components as well as behaviors in-between the two. For simplicity, we just consider these two possibilities.

As you can tell, sequential and random workloads will result in widely different performance characteristics from a disk. With sequential access, a disk operates in its most efficient mode, spending little time seeking and waiting for rotation and most of its time transferring data. With random access, just the opposite is true: most time is spent seeking and waiting for rotation and relatively little time is spent transferring data. To capture this difference in our analysis, we will assume that a disk can transfer data at  $S$  MB/s under a sequential workload, and  $R$  MB/s when under a random workload. In general,  $S$  is much greater than  $R$  (i.e.,  $S \gg R$ ).

To make sure we understand this difference, let's do a simple exercise. Specifically, let's calculate  $S$  and  $R$  given the following disk characteristics. Assume a sequential transfer of size 10 MB on average, and a random transfer of 10 KB on average. Also, assume the following disk characteristics:

Average seek time	7 ms
Average rotational delay	3 ms
Transfer rate of disk	50 MB/s

To compute  $S$ , we need to first figure out how time is spent in a typical 10 MB transfer. First, we spend 7 ms seeking, and then 3 ms rotating. Finally, transfer begins; 10 MB @ 50 MB/s leads to 1/5th of a second, or 200 ms, spent in transfer. Thus, for each 10 MB request, we spend 210 ms completing the request. To compute  $S$ , we just need to divide:

$$S = \frac{\text{Amount of Data}}{\text{Time to access}} = \frac{10 \text{ MB}}{210 \text{ ms}} = 47.62 \text{ MB/s}$$

As we can see, because of the large time spent transferring data,  $S$  is very near the peak bandwidth of the disk (the seek and rotational costs have been amortized).

We can compute  $R$  similarly. Seek and rotation are the same; we then compute the time spent in transfer, which is 10 KB @ 50 MB/s, or 0.195 ms.

$$R = \frac{\text{Amount of Data}}{\text{Time to access}} = \frac{10 \text{ KB}}{0.195 \text{ ms}} = 0.981 \text{ MB/s}$$

As we can see,  $R$  is less than 1 MB/s, and  $S/R$  is almost 50.

## Back To RAID-0 Analysis, Again

Let's now evaluate the performance of striping. As we said above, it is generally good. From a latency perspective, for example, the latency of a single-block request should be just about identical to that of a single disk; after all, RAID-0 will simply redirect that request to one of its disks.

From the perspective of steady-state throughput, we'd expect to get the full bandwidth of the system. Thus, throughput equals  $N$  (the number of disks) multiplied by  $S$  (the sequential bandwidth of a single disk). For a large number of random I/Os, we can again use all of the disks, and thus obtain  $N \cdot R$  MB/s. As we will see below, these values are both the simplest to calculate and will serve as an upper bound in comparison with other RAID levels.

### 38.5 RAID Level 1: Mirroring

Our first RAID level beyond striping is known as RAID level 1, or mirroring. With a mirrored system, we simply make more than one copy of each block in the system; each copy should be placed on a separate disk, of course. By doing so, we can tolerate disk failures.

In a typical mirrored system, we will assume that for each logical block, the RAID keeps two physical copies of it. Here is an example:

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Figure 38.3: Simple RAID-1: Mirroring

In the example, disk 0 and disk 1 have identical contents, and disk 2 and disk 3 do as well; the data is striped across these mirror pairs. In fact, you may have noticed that there are a number of different ways to place block copies across the disks. The arrangement above is a common one and is sometimes called **RAID-10** or (**RAID 1+0**) because it uses mirrored pairs (RAID-1) and then stripes (RAID-0) on top of them; another common arrangement is **RAID-01** (or **RAID 0+1**), which contains two large striping (RAID-0) arrays, and then mirrors (RAID-1) on top of them. For now, we will just talk about mirroring assuming the above layout.

When reading a block from a mirrored array, the RAID has a choice: it can read either copy. For example, if a read to logical block 5 is issued to the RAID, it is free to read it from either disk 2 or disk 3. When writing a block, though, no such choice exists: the RAID must update *both* copies of the data, in order to preserve reliability. Do note, though, that these writes can take place in parallel; for example, a write to logical block 5 could proceed to disks 2 and 3 at the same time.

### RAID-1 Analysis

Let us assess RAID-1. From a capacity standpoint, RAID-1 is expensive; with the mirroring level = 2, we only obtain half of our peak useful capacity. With  $N$  disks of  $B$  blocks, RAID-1 useful capacity is  $(N \cdot B)/2$ .

From a reliability standpoint, RAID-1 does well. It can tolerate the failure of any one disk. You may also notice RAID-1 can actually do better than this, with a little luck. Imagine, in the figure above, that disk 0 and disk 2 both failed. In such a situation, there is no data loss! More generally, a mirrored system (with mirroring level of 2) can tolerate 1 disk failure for certain, and up to  $N/2$  failures depending on which disks fail. In practice, we generally don't like to leave things like this to chance; thus most people consider mirroring to be good for handling a single failure.

Finally, we analyze performance. From the perspective of the latency of a single read request, we can see it is the same as the latency on a single disk; all the RAID-1 does is direct the read to one of its copies. A write is a little different: it requires two physical writes to complete before it is done. These two writes happen in parallel, and thus the time will be roughly equivalent to the time of a single write; however, because the logical write must wait for both physical writes to complete, it suffers the worst-case seek and rotational delay of the two requests, and thus (on average) will be slightly higher than a write to a single disk.

**ASIDE: THE RAID CONSISTENT-UPDATE PROBLEM**

Before analyzing RAID-1, let us first discuss a problem that arises in any multi-disk RAID system, known as the **consistent-update problem** [DAA05]. The problem occurs on a write to any RAID that has to update multiple disks during a single logical operation. In this case, let us assume we are considering a mirrored disk array.

Imagine the write is issued to the RAID, and then the RAID decides that it must be written to two disks, disk 0 and disk 1. The RAID then issues the write to disk 0, but just before the RAID can issue the request to disk 1, a power loss (or system crash) occurs. In this unfortunate case, let us assume that the request to disk 0 completed (but clearly the request to disk 1 did not, as it was never issued).

The result of this untimely power loss is that the two copies of the block are now **inconsistent**; the copy on disk 0 is the new version, and the copy on disk 1 is the old. What we would like to happen is for the state of both disks to change **atomically**, i.e., either both should end up as the new version or neither.

The general way to solve this problem is to use a **write-ahead log** of some kind to first record what the RAID is about to do (i.e., update two disks with a certain piece of data) before doing it. By taking this approach, we can ensure that in the presence of a crash, the right thing will happen; by running a **recovery** procedure that replays all pending transactions to the RAID, we can ensure that no two mirrored copies (in the RAID-1 case) are out of sync.

One last note: because logging to disk on every write is prohibitively expensive, most RAID hardware includes a small amount of non-volatile RAM (e.g., battery-backed) where it performs this type of logging. Thus, consistent update is provided without the high cost of logging to disk.

To analyze steady-state throughput, let us start with the sequential workload. When writing out to disk sequentially, each logical write must result in two physical writes; for example, when we write logical block 0 (in the figure above), the RAID internally would write it to both disk 0 and disk 1. Thus, we can conclude that the maximum bandwidth obtained during sequential writing to a mirrored array is  $(\frac{N}{2} \cdot S)$ , or half the peak bandwidth.

Unfortunately, we obtain the exact same performance during a sequential read. One might think that a sequential read could do better, because it only needs to read one copy of the data, not both. However, let's use an example to illustrate why this doesn't help much. Imagine we need to read blocks 0, 1, 2, 3, 4, 5, 6, and 7. Let's say we issue the read of 0 to disk 0, the read of 1 to disk 2, the read of 2 to disk 1, and the read of 3 to disk 3. We continue by issuing reads to 4, 5, 6, and 7 to disks 0, 2, 1, and 3, respectively. One might naively think that because we are utilizing all disks, we are achieving the full bandwidth of the array.

To see that this is not (necessarily) the case, however, consider the

requests a single disk receives (say disk 0). First, it gets a request for block 0; then, it gets a request for block 4 (skipping block 2). In fact, each disk receives a request for every other block. While it is rotating over the skipped block, it is not delivering useful bandwidth to the client. Thus, each disk will only deliver half its peak bandwidth. And thus, the sequential read will only obtain a bandwidth of  $(\frac{N}{2} \cdot S)$  MB/s.

Random reads are the best case for a mirrored RAID. In this case, we can distribute the reads across all the disks, and thus obtain the full possible bandwidth. Thus, for random reads, RAID-1 delivers  $N \cdot R$  MB/s.

Finally, random writes perform as you might expect:  $\frac{N}{2} \cdot R$  MB/s. Each logical write must turn into two physical writes, and thus while all the disks will be in use, the client will only perceive this as half the available bandwidth. Even though a write to logical block  $x$  turns into two parallel writes to two different physical disks, the bandwidth of many small requests only achieves half of what we saw with striping. As we will soon see, getting half the available bandwidth is actually pretty good!

### 38.6 RAID Level 4: Saving Space With Parity

We now present a different method of adding redundancy to a disk array known as **parity**. Parity-based approaches attempt to use less capacity and thus overcome the huge space penalty paid by mirrored systems. They do so at a cost, however: performance.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Figure 38.4: RAID-4 with Parity

Here is an example five-disk RAID-4 system (Figure 38.4). For each stripe of data, we have added a single **parity** block that stores the redundant information for that stripe of blocks. For example, parity block P1 has redundant information that it calculated from blocks 4, 5, 6, and 7.

To compute parity, we need to use a mathematical function that enables us to withstand the loss of any one block from our stripe. It turns out the simple function **XOR** does the trick quite nicely. For a given set of bits, the XOR of all of those bits returns a 0 if there are an even number of 1's in the bits, and a 1 if there are an odd number of 1's. For example:

C0	C1	C2	C3	P
0	0	1	1	XOR(0,0,1,1) = 0
0	1	0	0	XOR(0,1,0,0) = 1

In the first row (0,0,1,1), there are two 1's (C2, C3), and thus XOR of all of those values will be 0 (P); similarly, in the second row there is only one 1 (C1), and thus the XOR must be 1 (P). You can remember this in a simple way: that the number of 1s in any row must be an even (not odd) number; that is the **invariant** that the RAID must maintain in order for parity to be correct.

From the example above, you might also be able to guess how parity information can be used to recover from a failure. Imagine the column labeled C2 is lost. To figure out what values must have been in the column, we simply have to read in all the other values in that row (including the XOR'd parity bit) and **reconstruct** the right answer. Specifically, assume the first row's value in column C2 is lost (it is a 1); by reading the other values in that row (0 from C0, 0 from C1, 1 from C3, and 0 from the parity column P), we get the values 0, 0, 1, and 0. Because we know that XOR keeps an even number of 1's in each row, we know what the missing data must be: a 1. And that is how reconstruction works in a XOR-based parity scheme! Note also how we compute the reconstructed value: we just XOR the data bits and the parity bits together, in the same way that we calculated the parity in the first place.

Now you might be wondering: we are talking about XORing all of these bits, and yet from above we know that the RAID places 4KB (or larger) blocks on each disk; how do we apply XOR to a bunch of blocks to compute the parity? It turns out this is easy as well. Simply perform a bitwise XOR across each bit of the data blocks; put the result of each bitwise XOR into the corresponding bit slot in the parity block. For example, if we had blocks of size 4 bits (yes, this is still quite a bit smaller than a 4KB block, but you get the picture), they might look something like this:

Block0	Block1	Block2	Block3	Parity
00	10	11	10	11
10	01	00	01	10

As you can see from the figure, the parity is computed for each bit of each block and the result placed in the parity block.

## RAID-4 Analysis

Let us now analyze RAID-4. From a capacity standpoint, RAID-4 uses 1 disk for parity information for every group of disks it is protecting. Thus, our useful capacity for a RAID group is  $(N - 1) \cdot B$ .

Reliability is also quite easy to understand: RAID-4 tolerates 1 disk failure and no more. If more than one disk is lost, there is simply no way to reconstruct the lost data.

Finally, there is performance. This time, let us start by analyzing steady-state throughput. Sequential read performance can utilize all of the disks except for the parity disk, and thus deliver a peak effective bandwidth of  $(N - 1) \cdot S$  MB/s (an easy case).

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Figure 38.5: Full-stripe Writes In RAID-4

To understand the performance of sequential writes, we must first understand how they are done. When writing a big chunk of data to disk, RAID-4 can perform a simple optimization known as a **full-stripe write**. For example, imagine the case where the blocks 0, 1, 2, and 3 have been sent to the RAID as part of a write request (Figure 38.5).

In this case, the RAID can simply calculate the new value of P0 (by performing an XOR across the blocks 0, 1, 2, and 3) and then write all of the blocks (including the parity block) to the five disks above in parallel (highlighted in gray in the figure). Thus, full-stripe writes are the most efficient way for RAID-4 to write to disk.

Once we understand the full-stripe write, calculating the performance of sequential writes on RAID-4 is easy; the effective bandwidth is also  $(N - 1) \cdot S$  MB/s. Even though the parity disk is constantly in use during the operation, the client does not gain performance advantage from it.

Now let us analyze the performance of random reads. As you can also see from the figure above, a set of 1-block random reads will be spread across the data disks of the system but not the parity disk. Thus, the effective performance is:  $(N - 1) \cdot R$  MB/s.

Random writes, which we have saved for last, present the most interesting case for RAID-4. Imagine we wish to overwrite block 1 in the example above. We could just go ahead and overwrite it, but that would leave us with a problem: the parity block P0 would no longer accurately reflect the correct parity value of the stripe; in this example, P0 must also be updated. How can we update it both correctly and efficiently?

It turns out there are two methods. The first, known as **additive parity**, requires us to do the following. To compute the value of the new parity block, read in all of the other data blocks in the stripe in parallel (in the example, blocks 0, 2, and 3) and XOR those with the new block (1). The result is your new parity block. To complete the write, you can then write the new data and new parity to their respective disks, also in parallel.

The problem with this technique is that it scales with the number of disks, and thus in larger RAIDs requires a high number of reads to compute parity. Thus, the **subtractive parity** method.

For example, imagine this string of bits (4 data bits, one parity):

$$\begin{array}{ccccc} C_0 & C_1 & C_2 & C_3 & P \\ \hline 0 & 0 & 1 & 1 & \text{XOR}(0,0,1,1) = 0 \end{array}$$

Let's imagine that we wish to overwrite bit C2 with a new value which we will call  $C_{2,new}$ . The subtractive method works in three steps. First, we read in the old data at C2 ( $C_{2,old} = 1$ ) and the old parity ( $P_{old} = 0$ ).

Then, we compare the old data and the new data; if they are the same (e.g.,  $C_{new} = C_{old}$ ), then we know the parity bit will also remain the same (i.e.,  $P_{new} = P_{old}$ ). If, however, they are different, then we must flip the old parity bit to the opposite of its current state, that is, if ( $P_{old} == 1$ ),  $P_{new}$  will be set to 0; if ( $P_{old} == 0$ ),  $P_{new}$  will be set to 1. We can express this whole mess neatly with XOR (where  $\oplus$  is the XOR operator):

$$P_{new} = (C_{old} \oplus C_{new}) \oplus P_{old} \quad (38.1)$$

Because we are dealing with blocks, not bits, we perform this calculation over all the bits in the block (e.g., 4096 bytes in each block multiplied by 8 bits per byte). Thus, in most cases, the new block will be different than the old block and thus the new parity block will too.

You should now be able to figure out when we would use the additive parity calculation and when we would use the subtractive method. Think about how many disks would need to be in the system so that the additive method performs fewer I/Os than the subtractive method; what is the cross-over point?

For this performance analysis, let us assume we are using the subtractive method. Thus, for each write, the RAID has to perform 4 physical I/Os (two reads and two writes). Now imagine there are lots of writes submitted to the RAID; how many can RAID-4 perform in parallel? To understand, let us again look at the RAID-4 layout (Figure 38.6).

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
*4	5	6	7	+P1
8	9	10	11	P2
12	*13	14	15	+P3

Figure 38.6: Example: Writes To 4, 13, And Respective Parity Blocks

Now imagine there were 2 small writes submitted to the RAID-4 at about the same time, to blocks 4 and 13 (marked with \* in the diagram). The data for those disks is on disks 0 and 1, and thus the read and write to data could happen in parallel, which is good. The problem that arises is with the parity disk; both the requests have to read the related parity blocks for 4 and 13, parity blocks 1 and 3 (marked with +). Hopefully, the issue is now clear: the parity disk is a bottleneck under this type of workload; we sometimes thus call this the **small-write problem** for parity-based RAIDs. Thus, even though the data disks could be accessed in parallel, the parity disk prevents any parallelism from materializing; all writes to the system will be serialized because of the parity disk. Because the parity disk has to perform two I/Os (one read, one write) per logical I/O, we can compute the performance of small random writes in RAID-4 by computing the parity disk's performance on those two I/Os, and thus we achieve  $(R/2)$  MB/s. RAID-4 throughput under random small writes is terrible; it does not improve as you add disks to the system.

We conclude by analyzing I/O latency in RAID-4. As you now know, a single read (assuming no failure) is just mapped to a single disk, and thus its latency is equivalent to the latency of a single disk request. The latency of a single write requires two reads and then two writes; the reads can happen in parallel, as can the writes, and thus total latency is about twice that of a single disk (with some differences because we have to wait for both reads to complete and thus get the worst-case positioning time, but then the updates don't incur seek cost and thus may be a better-than-average positioning cost).

### 38.7 RAID Level 5: Rotating Parity

To address the small-write problem (at least, partially), Patterson, Gibson, and Katz introduced RAID-5. RAID-5 works almost identically to RAID-4, except that it **rotates** the parity block across drives (Figure 38.7).

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

Figure 38.7: RAID-5 With Rotated Parity

As you can see, the parity block for each stripe is now rotated across the disks, in order to remove the parity-disk bottleneck for RAID-4.

### RAID-5 Analysis

Much of the analysis for RAID-5 is identical to RAID-4. For example, the effective capacity and failure tolerance of the two levels are identical. So are sequential read and write performance. The latency of a single request (whether a read or a write) is also the same as RAID-4.

Random read performance is a little better, because we can now utilize all disks. Finally, random write performance improves noticeably over RAID-4, as it allows for parallelism across requests. Imagine a write to block 1 and a write to block 10; this will turn into requests to disk 1 and disk 4 (for block 1 and its parity) and requests to disk 0 and disk 2 (for block 10 and its parity). Thus, they can proceed in parallel. In fact, we can generally assume that given a large number of random requests, we will be able to keep all the disks about evenly busy. If that is the case, then our total bandwidth for small writes will be  $\frac{N}{4} \cdot R$  MB/s. The factor of four loss is due to the fact that each RAID-5 write still generates 4 total I/O operations, which is simply the cost of using parity-based RAID.

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	$N \cdot B$	$(N \cdot B)/2$	$(N - 1) \cdot B$	$(N - 1) \cdot B$
Reliability	0	1 (for sure) $\frac{N}{2}$ (if lucky)	1	1
Throughput				
Sequential Read	$N \cdot S$	$(N/2) \cdot S$	$(N - 1) \cdot S$	$(N - 1) \cdot S$
Sequential Write	$N \cdot S$	$(N/2) \cdot S$	$(N - 1) \cdot S$	$(N - 1) \cdot S$
Random Read	$N \cdot R$	$N \cdot R$	$(N - 1) \cdot R$	$N \cdot R$
Random Write	$N \cdot R$	$(N/2) \cdot R$	$\frac{1}{2} \cdot R$	$\frac{N}{4} R$
Latency				
Read	$T$	$T$	$T$	$T$
Write	$T$	$T$	$2T$	$2T$

Figure 38.8: RAID Capacity, Reliability, and Performance

Because RAID-5 is basically identical to RAID-4 except in the few cases where it is better, it has almost completely replaced RAID-4 in the marketplace. The only place where it has not is in systems that know they will never perform anything other than a large write, thus avoiding the small-write problem altogether [HLM94]; in those cases, RAID-4 is sometimes used as it is slightly simpler to build.

## 38.8 RAID Comparison: A Summary

We now summarize our simplified comparison of RAID levels in Figure 38.8. Note that we have omitted a number of details to simplify our analysis. For example, when writing in a mirrored system, the average seek time is a little higher than when writing to just a single disk, because the seek time is the max of two seeks (one on each disk). Thus, random write performance to two disks will generally be a little less than random write performance of a single disk. Also, when updating the parity disk in RAID-4/5, the first read of the old parity will likely cause a full seek and rotation, but the second write of the parity will only result in rotation.

However, the comparison in Figure 38.8 does capture the essential differences, and is useful for understanding tradeoffs across RAID levels. For the latency analysis, we simply use  $T$  to represent the time that a request to a single disk would take.

To conclude, if you strictly want performance and do not care about reliability, striping is obviously best. If, however, you want random I/O performance and reliability, mirroring is the best; the cost you pay is in lost capacity. If capacity and reliability are your main goals, then RAID-5 is the winner; the cost you pay is in small-write performance. Finally, if you are always doing sequential I/O and want to maximize capacity, RAID-5 also makes the most sense.

### 38.9 Other Interesting RAID Issues

There are a number of other interesting ideas that one could (and perhaps should) discuss when thinking about RAID. Here are some things we might eventually write about.

For example, there are many other RAID designs, including Levels 2 and 3 from the original taxonomy, and Level 6 to tolerate multiple disk faults [C+04]. There is also what the RAID does when a disk fails; sometimes it has a **hot spare** sitting around to fill in for the failed disk. What happens to performance under failure, and performance during reconstruction of the failed disk? There are also more realistic fault models, to take into account **latent sector errors** or **block corruption** [B+08], and lots of techniques to handle such faults (see the data integrity chapter for details). Finally, you can even build RAID as a software layer: such **software RAID** systems are cheaper but have other problems, including the consistent-update problem [DAA05].

### 38.10 Summary

We have discussed RAID. RAID transforms a number of independent disks into a large, more capacious, and more reliable single entity; importantly, it does so transparently, and thus hardware and software above is relatively oblivious to the change.

There are many possible RAID levels to choose from, and the exact RAID level to use depends heavily on what is important to the end-user. For example, mirrored RAID is simple, reliable, and generally provides good performance but at a high capacity cost. RAID-5, in contrast, is reliable and better from a capacity standpoint, but performs quite poorly when there are small writes in the workload. Picking a RAID and setting its parameters (chunk size, number of disks, etc.) properly for a particular workload is challenging, and remains more of an art than a science.

## References

- [B+08] "An Analysis of Data Corruption in the Storage Stack"  
 Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau  
 FAST '08, San Jose, CA, February 2008  
*Our own work analyzing how often disks actually corrupt your data. Not often, but sometimes! And thus something a reliable storage system must consider.*
- [BJ88] "Disk Shadowing"  
 D. Bitton and J. Gray  
 VLDB 1988  
*One of the first papers to discuss mirroring, herein called "shadowing".*
- [CL95] "Striping in a RAID level 5 disk array"  
 Peter M. Chen, Edward K. Lee  
 SIGMETRICS 1995  
*A nice analysis of some of the important parameters in a RAID-5 disk array.*
- [C+04] "Row-Diagonal Parity for Double Disk Failure Correction"  
 P. Corbett, B. English, A. Goel, T. Gracanac, S. Kleiman, J. Leong, S. Sankar  
 FAST '04, February 2004  
*Though not the first paper on a RAID system with two disks for parity, it is a recent and highly-understandable version of said idea. Read it to learn more.*
- [DAA05] "Journal-guided Resynchronization for Software RAID"  
 Timothy E. Denehy, A. Arpaci-Dusseau, R. Arpaci-Dusseau  
 FAST 2005  
*Our own work on the consistent-update problem. Here we solve it for Software RAID by integrating the journaling machinery of the file system above with the software RAID beneath it.*
- [HLM94] "File System Design for an NFS File Server Appliance"  
 Dave Hitz, James Lau, Michael Malcolm  
 USENIX Winter 1994, San Francisco, California, 1994  
*The sparse paper introducing a landmark product in storage, the write-anywhere file layout or WAFL file system that underlies the NetApp file server.*
- [K86] "Synchronized Disk Interleaving"  
 M.Y. Kim.  
 IEEE Transactions on Computers, Volume C-35: 11, November 1986  
*Some of the earliest work on RAID is found here.*
- [K88] "Small Disk Arrays - The Emerging Approach to High Performance"  
 F. Kurzweil.  
 Presentation at Spring COMPCON '88, March 1, 1988, San Francisco, California  
*Another early RAID reference.*
- [P+88] "Redundant Arrays of Inexpensive Disks"  
 D. Patterson, G. Gibson, R. Katz.  
 SIGMOD 1988  
*This is considered the RAID paper, written by famous authors Patterson, Gibson, and Katz. The paper has since won many test-of-time awards and ushered in the RAID era, including the name RAID itself!*

[PB86] "Providing Fault Tolerance in Parallel Secondary Storage Systems"

A. Park and K. Balasubramaniam

Department of Computer Science, Princeton, CS-TR-057-86, November 1986

*Another early work on RAID.*

[SG86] "Disk Striping"

K. Salem and H. Garcia-Molina.

IEEE International Conference on Data Engineering, 1986

*And yes, another early RAID work. There are a lot of these, which kind of came out of the woodwork when the RAID paper was published in SIGMOD.*

[S84] "Byzantine Generals in Action: Implementing Fail-Stop Processors"

F.B. Schneider.

ACM Transactions on Computer Systems, 2(2):145154, May 1984

*Finally, a paper that is not about RAID! This paper is actually about how systems fail, and how to make something behave in a fail-stop manner.*

## Homework

This section introduces `raid.py`, a simple RAID simulator you can use to shore up your knowledge of how RAID systems work. See the README for details.

### Questions

1. Use the simulator to perform some basic RAID mapping tests. Run with different levels (0, 1, 4, 5) and see if you can figure out the mappings of a set of requests. For RAID-5, see if you can figure out the difference between left-symmetric and left-asymmetric layouts. Use some different random seeds to generate different problems than above.
2. Do the same as the first problem, but this time vary the chunk size with `-C`. How does chunk size change the mappings?
3. Do the same as above, but use the `-r` flag to reverse the nature of each problem.
4. Now use the reverse flag but increase the size of each request with the `-S` flag. Try specifying sizes of 8k, 12k, and 16k, while varying the RAID level. What happens to the underlying I/O pattern when the size of the request increases? Make sure to try this with the sequential workload too (`-W sequential`); for what request sizes are RAID-4 and RAID-5 much more I/O efficient?
5. Use the timing mode of the simulator (`-t`) to estimate the performance of 100 random reads to the RAID, while varying the RAID levels, using 4 disks.
6. Do the same as above, but increase the number of disks. How does the performance of each RAID level scale as the number of disks increases?
7. Do the same as above, but use all writes (`-w 100`) instead of reads. How does the performance of each RAID level scale now? Can you do a rough estimate of the time it will take to complete the workload of 100 random writes?
8. Run the timing mode one last time, but this time with a sequential workload (`-W sequential`). How does the performance vary with RAID level, and when doing reads versus writes? How about when varying the size of each request? What size should you write to a RAID when using RAID-4 or RAID-5?

# Dynamically Loadable Kernel Modules

## Introduction

We often design systems to provide a common framework, but that expect problem-specific implementations to be provided at a later time. This approach is embraced by several standard design patterns (e.g. [Strategy](#), [Factory](#), [Plug-In](#)). These approaches have a few key elements in common:

- all implementations provide similar functionality in accordance with a common *interface*.
- the selection of a particular implementation can be deferred until run time.
- decoupling the overarching service from the plug-in implementations makes it possible for a system to work with a potentially open-ended set of algorithms or adaptors.

In most programs the set of available implementations is locked in at build-time. But many systems have the ability to select and load new implementations at run time as *dynamically loadable modules*. We see this with browser plug-ins. Operating systems may also support many different types of dynamically loadable modules (e.g. file systems, network protocols, device drivers). Device drivers are a particularly important and rich class of dynamically loadable modules ... and therefore a good example to study.

There are several compelling reasons for wanting device drivers to be dynamically loadable:

- the number of possible I/O devices is far too large to build all of them into the operating system.
- if we want an operating system to automatically work on any computer, it must have the ability to automatically identify and load the required device drivers.
- many devices (e.g. USB) are hot-pluggable, and we cannot know what drivers are required until the associated device is plugged in to the computer.
- new devices become available long after the operating system has been shipped, and so must be *after market* addable.
- most device drivers are developed by the hardware manufacturers, and delivered to customers independently from the operating system.

## Choosing Which Module to Load

In the abstract, a program needs an implementation and calls a *Factory* to obtain it. But how does the Factory know what implementation class to instantiate?

- for a browser plugin, there might be a MIME-type associated with the data to be handled, and the browser can consult a registry to find the plug-in associated with that MIME-type. This is a very general mechanism ... but it presumes that data is tagged with a type, and that somebody is maintaining a tag-to-plugin registry.
- at the other extreme, the Factory could load all of the known plug-ins and call a *probe* method in each to see which (if any) of the plug-ins could identify the data and claim responsibility for handling it.

Long ago, dynamically loadable device drivers used the probing process, but this was both unreliable (might incorrectly accept the wrong device) and dangerous (touching random registers in random devices). Today most I/O busses support self-identifying devices. Each device has type, model, and even serial number information that can be queried in a standard way (e.g. by *walking the configuration space*). This information can be used, in combination with a device driver registry, to automatically select a driver for a given device. These registries may support precedence rules that can choose the best from among multiple competing drivers (e.g. a generic VGA driver, a GeForce driver, and a GeForce GTX 980 driver).

# Loading a New Module

In many cases, the module to be loaded may be entirely self-contained (it makes no calls outside of itself) or uses only standard shared libraries (which are expected to be mapped in to the address space at well known locations). In these cases loading a new module is as simple as allocating some memory (or address space) and reading the new module into it.

In many cases (including device drivers and file systems) the dynamically loaded module may need to make use of other functions (e.g. memory allocation, synchronization, I/O) in the program into which it is being added. This means that the module to be loaded will (like an object module) have unresolved external references, and requires a *run-time loader* (a simplified linkage editor) that can look up and adjust all of those references as the new module is being loaded.

Note, however, that these references can only be from the dynamically loaded module into the program into which it is loaded (e.g. the operating system). The main program can never have any direct references into the dynamically loaded module ... because the dynamically loaded module may not always be there.

## Initialization and Registration

If the operating system is not allowed to have any direct references into a dynamically loadable module, how can the new module be used? When the run-time loader is invoked to load a new dynamically loadable module, it is common for it to return a vector that contains a pointer to at least one method: an initialization function.

After the module has been loaded into memory, the main program calls its initialization method. For a dynamically loaded device driver, the initialization method might:

- allocate memory and initialize driver data structures.
- allocate I/O resources (e.g. bus addresses, interrupt levels, DMA channels) and assign them to the devices to be managed.
- register all of the device instances it supports. Part of this registration would involve providing a vector (of pointers to standard device driver entry points) that client software could call in order to use these device instances.

Device instance configuration and initialization is another area where self-identifying devices have made it much easier to implement dynamically loaded device drivers:

- Long ago, devices were configured for particular bus addresses and interrupt levels with mechanical switches, that would be set before the card was plugged in to the bus. These resource assignments would be recorded in a system configuration table, which would be compiled (or read during system start-up) into the operating system, and used to select and configure corresponding device driver instances.
- More contemporary busses (like PCIe or USB) provide mechanisms to discover all of the available devices and learn what resources (e.g. bus addresses, DMA channels, interrupt levels) they require. The device driver can then allocate these resources from the associated bus driver, and assign the required resources to each device.

## Using a Dynamically Loaded Module

The operating system will provide some means by which processes can open device instances. In Linux the OS exports a pseudo file system (`/dev`) that is full of *special files*, each one associated with a registered device instance. When a process attempts to open one of those special files, the operating system creates a reference from the open file instance to the registered device instance. From then on, when ever the process issues a

`read(2)`, `write(2)`, or `ioctl(2)` system call on that file descriptor, the operating system forwards that call to the appropriate device driver entry point.

A similar approach is used when higher level frameworks (e.g. terminal sessions, network protocols or file systems) are implemented on top of a device. Each of those services maintains open references to the underlying devices, and when they need to talk to the device (e.g. to queue an I/O request or send a packet) the OS forwards that call to the appropriate device driver entry point.

The system often maintains a table of all registered device instances and the associated device driver entry points for each standard operation. When ever a request is to be made for any device, the operating system can simply index into this table by a device identifier to look up the address of the entry point to be called. Such mechanisms for registering heterogenous implementations and forwarding requests to the correct entry point are often referred to as *federation frameworks* because they combine a set of independent implementations into a single unified framework.

## Unloading

When all open file descriptors into those devices have been closed and the driver is no-longer needed, the operating system can call an shut-down method that will cause the driver to:

- un-register itself as a device driver
- shut down the devices it had been managing
- return all allocated memory and I/O resources back to the operating system.

After which, the module can be safely unloaded and that memory freed as well.

## The Criticality of Stable Interfaces

All of this his completely dependent on stable and well specified interfaces:

- the set of entry-points for any class of device driver must be well defined, and all drivers must compatably implement all of the relevant interfaces.
- the set of functions within the main program (OS) that the dynamically loaded modules are allowed to call must be well defined, and the interfaces to each of those functions must be stable.

If one device driver did not implement a standard entry point in the standard way, clients of that device would not work. Some functionality may be optional, and it may be acceptable for a device driver to refuse some requests. But this may make the application responsible for dealing with version some incompatabilities.

If an operating system does not implement some standard service function (e.g. memory allocation) in the standard way, a device driver written to that interface standard may not work when loaded into the non-compliant operating system.

There is often a tension between the conflicting needs to support new hardware and software features while retaining compatibility with old device drivers.

## Hot-Pluggable Devices and Drivers

One of the major advantages of dynamically loadable modules is that they can be loaded at any time; not merely during start-up. Hot-plug busses (e.g. USB) can generate events whenever a device is added to, or removed from the bus. In many systems a hot-plug manager:

- subscribes to hot-plug events.

- when a new device is inserted, walks the configuration space to identify the new device, finds, and loads the appropriate driver.
- when a device is removed, finds the associated driver and calls its *removal* method to inform it that the device is no longer on the bus.

Hot-plugable busses often have multiple power levels, and a newly inserted device may receive only enough power to enable it to be queried and configured. When the driver is ready to start using the device, it can instruct the bus to fully power the device. Some hot-pluggable busses also have mechanical safety interlocks to prevent a device from being removed while it is still in use. In these cases the driver must shut down and release the device before it can be removed.

## Summary

This discussion has focused on dynamically loadable device drivers, but most of the issues (selecting the module to be loaded, dependencies of the loaded module on the host program, initializing and shutting down dynamically loaded modules, binding clients to dynamically loaded modules, and defining and managing the interfaces between the loaded and hosting modules) are applicable to a much wider range of dynamically loadable modules.

Stable and well standardized interfaces are critical to any such framework:

- the methods to be implemented by the dynamically loaded modules.
- the services that can be used by the dynamically loaded modules.

## Interlude: Files and Directories

Thus far we have seen the development of two key operating system abstractions: the process, which is a virtualization of the CPU, and the address space, which is a virtualization of memory. In tandem, these two abstractions allow a program to run as if it is in its own private, isolated world; as if it has its own processor (or processors); as if it has its own memory. This illusion makes programming the system much easier and thus is prevalent today not only on desktops and servers but increasingly on all programmable platforms including mobile phones and the like.

In this section, we add one more critical piece to the virtualization puzzle: **persistent storage**. A persistent-storage device, such as a classic **hard disk drive** or a more modern **solid-state storage device**, stores information permanently (or at least, for a long time). Unlike memory, whose contents are lost when there is a power loss, a persistent-storage device keeps such data intact. Thus, the OS must take extra care with such a device: this is where users keep data that they really care about.

### CRUX: HOW TO MANAGE A PERSISTENT DEVICE

How should the OS manage a persistent device? What are the APIs? What are the important aspects of the implementation?

Thus, in the next few chapters, we will explore critical techniques for managing persistent data, focusing on methods to improve performance and reliability. We begin, however, with an overview of the API: the interfaces you'll expect to see when interacting with a UNIX file system.

### 39.1 Files and Directories

Two key abstractions have developed over time in the virtualization of storage. The first is the **file**. A file is simply a linear array of bytes, each of which you can read or write. Each file has some kind of **low-level**

**name**, usually a number of some kind; often, the user is not aware of this name (as we will see). For historical reasons, the low-level name of a file is often referred to as its **inode number**. We'll be learning a lot more about inodes in future chapters; for now, just assume that each file has an inode number associated with it.

In most systems, the OS does not know much about the structure of the file (e.g., whether it is a picture, or a text file, or C code); rather, the responsibility of the file system is simply to store such data persistently on disk and make sure that when you request the data again, you get what you put there in the first place. Doing so is not as simple as it seems!

The second abstraction is that of a **directory**. A directory, like a file, also has a low-level name (i.e., an inode number), but its contents are quite specific: it contains a list of (user-readable name, low-level name) pairs. For example, let's say there is a file with the low-level name "10", and it is referred to by the user-readable name of "foo". The directory that "foo" resides in thus would have an entry ("foo", "10") that maps the user-readable name to the low-level name. Each entry in a directory refers to either files or other directories. By placing directories within other directories, users are able to build an arbitrary **directory tree** (or **directory hierarchy**), under which all files and directories are stored.

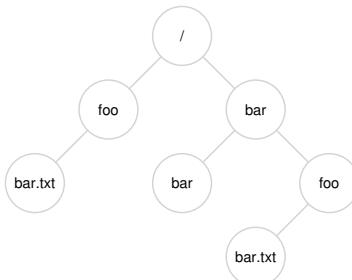


Figure 39.1: An Example Directory Tree

The directory hierarchy starts at a **root directory** (in UNIX-based systems, the root directory is simply referred to as `/`) and uses some kind of **separator** to name subsequent **sub-directories** until the desired file or directory is named. For example, if a user created a directory `foo` in the root directory `/`, and then created a file `bar.txt` in the directory `foo`, we could refer to the file by its **absolute pathname**, which in this case would be `/foo/bar.txt`. See Figure 39.1 for a more complex directory tree; valid directories in the example are `/`, `/foo`, `/bar`, `/bar/bar`, `/bar/foo` and valid files are `/foo/bar.txt` and `/bar/foo/bar.txt`. Directories and files can have the same name as long as they are in different locations in the file-system tree (e.g., there are two files named `bar.txt` in the figure, `/foo/bar.txt` and `/bar/foo/bar.txt`).

**TIP: THINK CAREFULLY ABOUT NAMING**

Naming is an important aspect of computer systems [SK09]. In UNIX systems, virtually everything that you can think of is named through the file system. Beyond just files, devices, pipes, and even processes [K84] can be found in what looks like a plain old file system. This uniformity of naming eases your conceptual model of the system, and makes the system simpler and more modular. Thus, whenever creating a system or interface, think carefully about what names you are using.

You may also notice that the file name in this example often has two parts: `bar` and `.txt`, separated by a period. The first part is an arbitrary name, whereas the second part of the file name is usually used to indicate the **type** of the file, e.g., whether it is C code (e.g., `.c`), or an image (e.g., `.jpg`), or a music file (e.g., `.mp3`). However, this is usually just a **convention**: there is usually no enforcement that the data contained in a file named `main.c` is indeed C source code.

Thus, we can see one great thing provided by the file system: a convenient way to **name** all the files we are interested in. Names are important in systems as the first step to accessing any resource is being able to name it. In UNIX systems, the file system thus provides a unified way to access files on disk, USB stick, CD-ROM, many other devices, and in fact many other things, all located under the single directory tree.

## 39.2 The File System Interface

Let's now discuss the file system interface in more detail. We'll start with the basics of creating, accessing, and deleting files. You may think this is straightforward, but along the way we'll discover the mysterious call that is used to remove files, known as `unlink()`. Hopefully, by the end of this chapter, this mystery won't be so mysterious to you!

## 39.3 Creating Files

We'll start with the most basic of operations: creating a file. This can be accomplished with the `open` system call; by calling `open()` and passing it the `O_CREAT` flag, a program can create a new file. Here is some example code to create a file called "foo" in the current working directory.

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
```

The routine `open()` takes a number of different flags. In this example, the second parameter creates the file (`O_CREAT`) if it does not exist, ensures that the file can only be written to (`O_WRONLY`), and, if the file already exists, truncates it to a size of zero bytes thus removing any existing content (`O_TRUNC`). The third parameter specifies permissions, in this case making the file readable and writable by the owner.

**ASIDE: THE `CREAT()` SYSTEM CALL**

The older way of creating a file is to call `creat()`, as follows:

```
int fd = creat("foo"); // option: add second flag to set permissions
```

You can think of `creat()` as `open()` with the following flags: `O_CREAT | O_WRONLY | O_TRUNC`. Because `open()` can create a file, the usage of `creat()` has somewhat fallen out of favor (indeed, it could just be implemented as a library call to `open()`); however, it does hold a special place in UNIX lore. Specifically, when Ken Thompson was asked what he would do differently if he were redesigning UNIX, he replied: “I’d spell `creat` with an e.”

One important aspect of `open()` is what it returns: a **file descriptor**. A file descriptor is just an integer, private per process, and is used in UNIX systems to access files; thus, once a file is opened, you use the file descriptor to read or write the file, assuming you have permission to do so. In this way, a file descriptor is a **capability** [L84], i.e., an opaque handle that gives you the power to perform certain operations. Another way to think of a file descriptor is as a pointer to an object of type `file`; once you have such an object, you can call other “methods” to access the file, like `read()` and `write()`. We’ll see just how a file descriptor is used below.

## 39.4 Reading and Writing Files

Once we have some files, of course we might like to read or write them. Let’s start by reading an existing file. If we were typing at a command line, we might just use the program `cat` to dump the contents of the file to the screen.

```
prompt> echo hello > foo
prompt> cat foo
hello
prompt>
```

In this code snippet, we redirect the output of the program `echo` to the file `foo`, which then contains the word “hello” in it. We then use `cat` to see the contents of the file. But how does the `cat` program access the file `foo`?

To find this out, we’ll use an incredibly useful tool to trace the system calls made by a program. On Linux, the tool is called `strace`; other systems have similar tools (see `dtruss` on Mac OS X, or `truss` on some older UNIX variants). What `strace` does is trace every system call made by a program while it runs, and dump the trace to the screen for you to see.

**TIP: USE STRACE (AND SIMILAR TOOLS)**

The `strace` tool provides an awesome way to see what programs are up to. By running it, you can trace which system calls a program makes, see the arguments and return codes, and generally get a very good idea of what is going on.

The tool also takes some arguments which can be quite useful. For example, `-f` follows any fork'd children too; `-t` reports the time of day at each call; `-e trace=open,close,read,write` only traces calls to those system calls and ignores all others. There are many more powerful flags — read the man pages and find out how to harness this wonderful tool.

Here is an example of using `strace` to figure out what `cat` is doing (some calls removed for readability):

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)      = 3
read(3, "hello\n", 4096)              = 6
write(1, "hello\n", 6)                = 6
hello
read(3, "", 4096)                   = 0
close(3)                            = 0
...
prompt>
```

The first thing that `cat` does is open the file for reading. A couple of things we should note about this; first, that the file is only opened for reading (not writing), as indicated by the `O_RDONLY` flag; second, that the 64-bit offset be used (`O_LARGEFILE`); third, that the call to `open()` succeeds and returns a file descriptor, which has the value of 3.

Why does the first call to `open()` return 3, not 0 or perhaps 1 as you might expect? As it turns out, each running process already has three files open, standard input (which the process can read to receive input), standard output (which the process can write to in order to dump information to the screen), and standard error (which the process can write error messages to). These are represented by file descriptors 0, 1, and 2, respectively. Thus, when you first open another file (as `cat` does above), it will almost certainly be file descriptor 3.

After the `open` succeeds, `cat` uses the `read()` system call to repeatedly read some bytes from a file. The first argument to `read()` is the file descriptor, thus telling the file system which file to read; a process can of course have multiple files open at once, and thus the descriptor enables the operating system to know which file a particular read refers to. The second argument points to a buffer where the result of the `read()` will be placed; in the system-call trace above, `strace` shows the results of the `read` in this spot ("hello"). The third argument is the size of the buffer, which

in this case is 4 KB. The call to `read()` returns successfully as well, here returning the number of bytes it read (6, which includes 5 for the letters in the word “hello” and one for an end-of-line marker).

At this point, you see another interesting result of the strace: a single call to the `write()` system call, to the file descriptor 1. As we mentioned above, this descriptor is known as the standard output, and thus is used to write the word “hello” to the screen as the program `cat` is meant to do. But does it call `write()` directly? Maybe (if it is highly optimized). But if not, what `cat` might do is call the library routine `printf()`; internally, `printf()` figures out all the formatting details passed to it, and eventually calls `write` on the standard output to print the results to the screen.

The `cat` program then tries to read more from the file, but since there are no bytes left in the file, the `read()` returns 0 and the program knows that this means it has read the entire file. Thus, the program calls `close()` to indicate that it is done with the file “foo”, passing in the corresponding file descriptor. The file is thus closed, and the reading of it thus complete.

Writing a file is accomplished via a similar set of steps. First, a file is opened for writing, then the `write()` system call is called, perhaps repeatedly for larger files, and then `close()`. Use `strace` to trace writes to a file, perhaps of a program you wrote yourself, or by tracing the `dd` utility, e.g., `dd if=foo of=bar`.

## 39.5 Reading And Writing, But Not Sequentially

Thus far, we’ve discussed how to read and write files, but all access has been **sequential**; that is, we have either read a file from the beginning to the end, or written a file out from beginning to end.

Sometimes, however, it is useful to be able to read or write to a specific offset within a file; for example, if you build an index over a text document, and use it to look up a specific word, you may end up reading from some **random** offsets within the document. To do so, we will use the `lseek()` system call. Here is the function prototype:

```
off_t lseek(int fildes, off_t offset, int whence);
```

The first argument is familiar (a file descriptor). The second argument is the `offset`, which positions the **file offset** to a particular location within the file. The third argument, called `whence` for historical reasons, determines exactly how the seek is performed. From the man page:

```
If whence is SEEK_SET, the offset is set to offset bytes.  
If whence is SEEK_CUR, the offset is set to its current  
location plus offset bytes.  
If whence is SEEK_END, the offset is set to the size of  
the file plus offset bytes.
```

As you can tell from this description, for each file a process opens, the OS tracks a “current” offset, which determines where the next read or

**ASIDE: CALLING `lseek()` DOES NOT PERFORM A DISK SEEK**

The poorly-named system call `lseek()` confuses many a student trying to understand disks and how the file systems atop them work. Do not confuse the two! The `lseek()` call simply changes a variable in OS memory that tracks, for a particular process, at which offset to which its next read or write will start. A disk seek occurs when a read or write issued to the disk is not on the same track as the last read or write, and thus necessitates a head movement. Making this even more confusing is the fact that calling `lseek()` to read or write from/to random parts of a file, and then reading/writing to those random parts, will indeed lead to more disk seeks. Thus, calling `lseek()` can certainly lead to a seek in an upcoming read or write, but absolutely does not cause any disk I/O to occur itself.

write will begin reading from or writing to within the file. Thus, part of the abstraction of an open file is that it has a current offset, which is updated in one of two ways. The first is when a read or write of  $N$  bytes takes place,  $N$  is added to the current offset; thus each read or write *implicitly* updates the offset. The second is *explicitly* with `lseek`, which changes the offset as specified above.

Note that this call `lseek()` has nothing to do with the **seek** operation of a disk, which moves the disk arm. The call to `lseek()` simply changes the value of a variable within the kernel; when the I/O is performed, depending on where the disk head is, the disk may or may not perform an actual seek to fulfill the request.

### 39.6 Writing Immediately with `fsync()`

Most times when a program calls `write()`, it is just telling the file system: please write this data to persistent storage, at some point in the future. The file system, for performance reasons, will **buffer** such writes in memory for some time (say 5 seconds, or 30); at that later point in time, the write(s) will actually be issued to the storage device. From the perspective of the calling application, writes seem to complete quickly, and only in rare cases (e.g., the machine crashes after the `write()` call but before the write to disk) will data be lost.

However, some applications require something more than this eventual guarantee. For example, in a database management system (DBMS), development of a correct recovery protocol requires the ability to force writes to disk from time to time.

To support these types of applications, most file systems provide some additional control APIs. In the UNIX world, the interface provided to applications is known as `fsync(int fd)`. When a process calls `fsync()` for a particular file descriptor, the file system responds by forcing all **dirty** (i.e., not yet written) data to disk, for the file referred to by the specified

file descriptor. The `fsync()` routine returns once all of these writes are complete.

Here is a simple example of how to use `fsync()`. The code opens the file `foo`, writes a single chunk of data to it, and then calls `fsync()` to ensure the writes are forced immediately to disk. Once the `fsync()` returns, the application can safely move on, knowing that the data has been persisted (if `fsync()` is correctly implemented, that is).

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
assert(fd > -1);
int rc = write(fd, buffer, size);
assert(rc == size);
rc = fsync(fd);
assert(rc == 0);
```

Interestingly, this sequence does not guarantee everything that you might expect; in some cases, you also need to `fsync()` the directory that contains the file `foo`. Adding this step ensures not only that the file itself is on disk, but that the file, if newly created, also is durably a part of the directory. Not surprisingly, this type of detail is often overlooked, leading to many application-level bugs [P+13].

### 39.7 Renaming Files

Once we have a file, it is sometimes useful to be able to give a file a different name. When typing at the command line, this is accomplished with `mv` command; in this example, the file `foo` is renamed `bar`:

```
prompt> mv foo bar
```

Using `strace`, we can see that `mv` uses the system call `rename(char *old, char *new)`, which takes precisely two arguments: the original name of the file (`old`) and the new name (`new`).

One interesting guarantee provided by the `rename()` call is that it is (usually) implemented as an **atomic** call with respect to system crashes; if the system crashes during the renaming, the file will either be named the old name or the new name, and no odd in-between state can arise. Thus, `rename()` is critical for supporting certain kinds of applications that require an atomic update to file state.

Let's be a little more specific here. Imagine that you are using a file editor (e.g., `emacs`), and you insert a line into the middle of a file. The file's name, for the example, is `foo.txt`. The way the editor might update the file to guarantee that the new file has the original contents plus the line inserted is as follows (ignoring error-checking for simplicity):

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC,
              S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

What the editor does in this example is simple: write out the new version of the file under a temporary name (`foo.txt.tmp`), force it to disk with `fsync()`, and then, when the application is certain the new file metadata and contents are on the disk, rename the temporary file to the original file's name. This last step atomically swaps the new file into place, while concurrently deleting the old version of the file, and thus an atomic file update is achieved.

## 39.8 Getting Information About Files

Beyond file access, we expect the file system to keep a fair amount of information about each file it is storing. We generally call such data about files **metadata**. To see the metadata for a certain file, we can use the `stat()` or `fstat()` system calls. These calls take a pathname (or file descriptor) to a file and fill in a `stat` structure as seen here:

```
struct stat {
    dev_t      st_dev;      /* ID of device containing file */
    ino_t      st_ino;      /* inode number */
    mode_t     st_mode;     /* protection */
    nlink_t    st_nlink;    /* number of hard links */
    uid_t      st_uid;      /* user ID of owner */
    gid_t      st_gid;      /* group ID of owner */
    dev_t      st_rdev;     /* device ID (if special file) */
    off_t      st_size;     /* total size, in bytes */
    blksize_t  st_blksize;  /* blocksize for filesystem I/O */
    blkcnt_t   st_blocks;   /* number of blocks allocated */
    time_t     st_atime;    /* time of last access */
    time_t     st_mtime;    /* time of last modification */
    time_t     st_ctime;    /* time of last status change */
};
```

You can see that there is a lot of information kept about each file, including its size (in bytes), its low-level name (i.e., inode number), some ownership information, and some information about when the file was accessed or modified, among other things. To see this information, you can use the command line tool `stat`:

```
prompt> echo hello > file
prompt> stat file
  File: 'file'
  Size: 6 Blocks: 8          IO Block: 4096   regular file
Device: 811h/2065d Inode: 67158084  Links: 1
Access: (0640/-rw-r-----)  Uid: (30686/ remzi) Gid: (30686/ remzi)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500
```

As it turns out, each file system usually keeps this type of information in a structure called an **inode**<sup>1</sup>. We'll be learning a lot more about inodes when we talk about file system implementation. For now, you should just think of an inode as a persistent data structure kept by the file system that has information like we see above inside of it.

### 39.9 Removing Files

At this point, we know how to create files and access them, either sequentially or not. But how do you delete files? If you've used UNIX, you probably think you know: just run the program `rm`. But what system call does `rm` use to remove a file?

Let's use our old friend `strace` again to find out. Here we remove that pesky file "foo":

```
prompt> strace rm foo
...
unlink("foo")                                = 0
...
```

We've removed a bunch of unrelated cruft from the traced output, leaving just a single call to the mysteriously-named system call `unlink()`. As you can see, `unlink()` just takes the name of the file to be removed, and returns zero upon success. But this leads us to a great puzzle: why is this system call named "unlink"? Why not just "remove" or "delete". To understand the answer to this puzzle, we must first understand more than just files, but also directories.

### 39.10 Making Directories

Beyond files, a set of directory-related system calls enable you to make, read, and delete directories. Note you can never write to a directory directly; because the format of the directory is considered file system metadata, you can only update a directory indirectly by, for example, creating files, directories, or other object types within it. In this way, the file system makes sure that the contents of the directory always are as expected.

To create a directory, a single system call, `mkdir()`, is available. The eponymous `mkdir` program can be used to create such a directory. Let's take a look at what happens when we run the `mkdir` program to make a simple directory called `foo`:

```
prompt> strace mkdir foo
...
mkdir("foo", 0777)                            = 0
...
prompt>
```

---

<sup>1</sup>Some file systems call these structures similar, but slightly different, names, such as `dnodes`; the basic idea is similar however.

**TIP: BE WARY OF POWERFUL COMMANDS**

The program `rm` provides us with a great example of powerful commands, and how sometimes too much power can be a bad thing. For example, to remove a bunch of files at once, you can type something like:

```
prompt> rm *
```

where the `*` will match all files in the current directory. But sometimes you want to also delete the directories too, and in fact all of their contents. You can do this by telling `rm` to recursively descend into each directory, and remove its contents too:

```
prompt> rm -rf *
```

Where you get into trouble with this small string of characters is when you issue the command, accidentally, from the root directory of a file system, thus removing every file and directory from it. Oops!

Thus, remember the double-edged sword of powerful commands; while they give you the ability to do a lot of work with a small number of keystrokes, they also can quickly and readily do a great deal of harm.

When such a directory is created, it is considered “empty”, although it does have a bare minimum of contents. Specifically, an empty directory has two entries: one entry that refers to itself, and one entry that refers to its parent. The former is referred to as the `“.”` (dot) directory, and the latter as `“..”` (dot-dot). You can see these directories by passing a flag (`-a`) to the program `ls`:

```
prompt> ls -a
./  ../
prompt> ls -al
total 8
drwxr-x---  2 remzi remzi   6 Apr 30 16:17 .
drwxr-x--- 26 remzi remzi 4096 Apr 30 16:17 ..
```

### 39.11 Reading Directories

Now that we’ve created a directory, we might wish to read one too. Indeed, that is exactly what the program `ls` does. Let’s write our own little tool like `ls` and see how it is done.

Instead of just opening a directory as if it were a file, we instead use a new set of calls. Below is an example program that prints the contents of a directory. The program uses three calls, `opendir()`, `readdir()`, and `closedir()`, to get the job done, and you can see how simple the interface is; we just use a simple loop to read one directory entry at a time, and print out the name and inode number of each file in the directory.

```

int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) {
        printf("%d %s\n", (int) d->d_ino, d->d_name);
    }
    closedir(dp);
    return 0;
}

```

The declaration below shows the information available within each directory entry in the `struct dirent` data structure:

```

struct dirent {
    char          d_name[256]; /* filename */
    ino_t         d_ino;      /* inode number */
    off_t         d_off;      /* offset to the next dirent */
    unsigned short d_reclen; /* length of this record */
    unsigned char d_type;   /* type of file */
};

```

Because directories are light on information (basically, just mapping the name to the inode number, along with a few other details), a program may want to call `stat()` on each file to get more information on each, such as its length or other detailed information. Indeed, this is exactly what `ls` does when you pass it the `-l` flag; try `strace` on `ls` with and without that flag to see for yourself.

### 39.12 Deleting Directories

Finally, you can delete a directory with a call to `rmdir()` (which is used by the program of the same name, `rmdir`). Unlike file deletion, however, removing directories is more dangerous, as you could potentially delete a large amount of data with a single command. Thus, `rmdir()` has the requirement that the directory be empty (i.e., only has `"."` and `".."` entries) before it is deleted. If you try to delete a non-empty directory, the call to `rmdir()` simply will fail.

### 39.13 Hard Links

We now come back to the mystery of why removing a file is performed via `unlink()`, by understanding a new way to make an entry in the file system tree, through a system call known as `link()`. The `link()` system call takes two arguments, an old pathname and a new one; when you “link” a new file name to an old one, you essentially create another way to refer to the same file. The command-line program `ln` is used to do this, as we see in this example:

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
```

Here we created a file with the word “hello” in it, and called the file `file`<sup>2</sup>. We then create a hard link to that file using the `ln` program. After this, we can examine the file by either opening `file` or `file2`.

The way `link` works is that it simply creates another name in the directory you are creating the link to, and refers it to the *same* inode number (i.e., low-level name) of the original file. The file is not copied in any way; rather, you now just have two human names (`file` and `file2`) that both refer to the same file. We can even see this in the directory itself, by printing out the inode number of each file:

```
prompt> ls -i file file2
67158084 file
67158084 file2
prompt>
```

By passing the `-i` flag to `ls`, it prints out the inode number of each file (as well as the file name). And thus you can see what `link` really has done: just make a new reference to the same exact inode number (67158084 in this example).

By now you might be starting to see why `unlink()` is called `unlink()`. When you create a file, you are really doing *two* things. First, you are making a structure (the inode) that will track virtually all relevant information about the file, including its size, where its blocks are on disk, and so forth. Second, you are *linking* a human-readable name to that file, and putting that link into a directory.

After creating a hard link to a file, to the file system, there is no difference between the original file name (`file`) and the newly created file name (`file2`); indeed, they are both just links to the underlying metadata about the file, which is found in inode number 67158084.

Thus, to remove a file from the file system, we call `unlink()`. In the example above, we could for example remove the file named `file`, and still access the file without difficulty:

```
prompt> rm file
removed 'file'
prompt> cat file2
hello
```

The reason this works is because when the file system unlinks `file`, it checks a **reference count** within the inode number. This reference count

---

<sup>2</sup>Note how creative the authors of this book are. We also used to have a cat named “Cat” (true story). However, she died, and we now have a hamster named “Hammy.” Update: Hammy is now dead too. The pet bodies are piling up.

(sometimes called the **link count**) allows the file system to track how many different file names have been linked to this particular inode. When `unlink()` is called, it removes the “link” between the human-readable name (the file that is being deleted) to the given inode number, and decrements the reference count; only when the reference count reaches zero does the file system also free the inode and related data blocks, and thus truly “delete” the file.

You can see the reference count of a file using `stat()` of course. Let’s see what it is when we create and delete hard links to a file. In this example, we’ll create three links to the same file, and then delete them. Watch the link count!

```

prompt> echo hello > file
prompt> stat file
... Inode: 67158084    Links: 1 ...
prompt> ln file file2
prompt> stat file
... Inode: 67158084    Links: 2 ...
prompt> stat file2
... Inode: 67158084    Links: 2 ...
prompt> ln file2 file3
prompt> stat file
... Inode: 67158084    Links: 3 ...
prompt> rm file
prompt> stat file2
... Inode: 67158084    Links: 2 ...
prompt> rm file2
prompt> stat file3
... Inode: 67158084    Links: 1 ...
prompt> rm file3

```

### 39.14 Symbolic Links

There is one other type of link that is really useful, and it is called a **symbolic link** or sometimes a **soft link**. As it turns out, hard links are somewhat limited: you can’t create one to a directory (for fear that you will create a cycle in the directory tree); you can’t hard link to files in other disk partitions (because inode numbers are only unique within a particular file system, not across file systems); etc. Thus, a new type of link called the symbolic link was created.

To create such a link, you can use the same program `ln`, but with the `-s` flag. Here is an example:

```

prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello

```

As you can see, creating a soft link looks much the same, and the original file can now be accessed through the file name `file` as well as the symbolic link name `file2`.

However, beyond this surface similarity, symbolic links are actually quite different from hard links. The first difference is that a symbolic link is actually a file itself, of a different type. We've already talked about regular files and directories; symbolic links are a third type the file system knows about. A stat on the symlink reveals all:

```
prompt> stat file  
... regular file ...  
prompt> stat file2  
... symbolic link ...
```

Running ls also reveals this fact. If you look closely at the first character of the long-form of the output from ls, you can see that the first character in the left-most column is a - for regular files, a d for directories, and an l for soft links. You can also see the size of the symbolic link (4 bytes in this case), as well as what the link points to (the file named file).

```
prompt> ls -al  
drwxr-x--- 2 remzi remzi 29 May 3 19:10 ./  
drwxr-x--- 27 remzi remzi 4096 May 3 15:14 ../  
-rw-r----- 1 remzi remzi 6 May 3 19:10 file  
lrwxrwxrwx 1 remzi remzi 4 May 3 19:10 file2 -> file
```

The reason that file2 is 4 bytes is because the way a symbolic link is formed is by holding the pathname of the linked-to file as the data of the link file. Because we've linked to a file named file, our link file file2 is small (4 bytes). If we link to a longer pathname, our link file would be bigger:

```
prompt> echo hello > alonergfilename  
prompt> ln -s alonergfilename file3  
prompt> ls -al alonergfilename file3  
-rw-r----- 1 remzi remzi 6 May 3 19:17 alonergfilename  
lrwxrwxrwx 1 remzi remzi 15 May 3 19:17 file3 -> alonergfilename
```

Finally, because of the way symbolic links are created, they leave the possibility for what is known as a **dangling reference**:

```
prompt> echo hello > file  
prompt> ln -s file file2  
prompt> cat file2  
hello  
prompt> rm file  
prompt> cat file2  
cat: file2: No such file or directory
```

As you can see in this example, quite unlike hard links, removing the original file named file causes the link to point to a pathname that no longer exists.

### 39.15 Making and Mounting a File System

We've now toured the basic interfaces to access files, directories, and certain types of special types of links. But there is one more topic we should discuss: how to assemble a full directory tree from many underlying file systems. This task is accomplished via first making file systems, and then mounting them to make their contents accessible.

To make a file system, most file systems provide a tool, usually referred to as `mkfs` (pronounced "make fs"), that performs exactly this task. The idea is as follows: give the tool, as input, a device (such as a disk partition, e.g., `/dev/sda1`) a file system type (e.g., `ext3`), and it simply writes an empty file system, starting with a root directory, onto that disk partition. And `mkfs` said, let there be a file system!

However, once such a file system is created, it needs to be made accessible within the uniform file-system tree. This task is achieved via the `mount` program (which makes the underlying system call `mount()` to do the real work). What `mount` does, quite simply is take an existing directory as a target **mount point** and essentially paste a new file system onto the directory tree at that point.

An example here might be useful. Imagine we have an unmounted `ext3` file system, stored in device partition `/dev/sda1`, that has the following contents: a root directory which contains two sub-directories, `a` and `b`, each of which in turn holds a single file named `foo`. Let's say we wish to mount this file system at the mount point `/home/users`. We would type something like this:

```
prompt> mount -t ext3 /dev/sda1 /home/users
```

If successful, the `mount` would thus make this new file system available. However, note how the new file system is now accessed. To look at the contents of the root directory, we would use `ls` like this:

```
prompt> ls /home/users/
a b
```

As you can see, the pathname `/home/users/` now refers to the root of the newly-mounted directory. Similarly, we could access files `a` and `b` with the pathnames `/home/users/a` and `/home/users/b`. Finally, the files named `foo` could be accessed via `/home/users/a/foo` and `/home/users/b/foo`. And thus the beauty of `mount`: instead of having a number of separate file systems, `mount` unifies all file systems into one tree, making naming uniform and convenient.

To see what is mounted on your system, and at which points, simply run the `mount` program. You'll see something like this:

```
/dev/sda1 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
/dev/sda5 on /tmp type ext3 (rw)
/dev/sda7 on /var/vice/cache type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
AFS on /afs type afs (rw)
```

This crazy mix shows that a whole number of different file systems, including ext3 (a standard disk-based file system), the proc file system (a file system for accessing information about current processes), tmpfs (a file system just for temporary files), and AFS (a distributed file system) are all glued together onto this one machine's file-system tree.

### 39.16 Summary

The file system interface in UNIX systems (and indeed, in any system) is seemingly quite rudimentary, but there is a lot to understand if you wish to master it. Nothing is better, of course, than simply using it (a lot). So please do so! Of course, read more; as always, Stevens [SR05] is the place to begin.

We've toured the basic interfaces, and hopefully understood a little bit about how they work. Even more interesting is how to implement a file system that meets the needs of the API, a topic we will delve into in great detail next.

## References

- [K84] "Processes as Files"  
Tom J. Killian  
USENIX, June 1984  
*The paper that introduced the /proc file system, where each process can be treated as a file within a pseudo file system. A clever idea that you can still see in modern UNIX systems.*
- [L84] "Capability-Based Computer Systems"  
Henry M. Levy  
Digital Press, 1984  
Available: <http://homes.cs.washington.edu/~levy/capabook>  
*An excellent overview of early capability-based systems.*
- [P+13] "Towards Efficient, Portable Application-Level Consistency"  
Thanumalayan S. Pillai, Vijay Chidambaran, Joo-Young Hwang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau  
HotDep '13, November 2013  
*Our own work that shows how readily applications can make mistakes in committing data to disk; in particular, assumptions about the file system creep into applications and thus make the applications work correctly only if they are running on a specific file system.*
- [SK09] "Principles of Computer System Design"  
Jerome H. Saltzer and M. Frans Kaashoek  
Morgan-Kaufmann, 2009  
*This tour de force of systems is a must-read for anybody interested in the field. It's how they teach systems at MIT. Read it once, and then read it a few more times to let it all soak in.*
- [SR05] "Advanced Programming in the UNIX Environment"  
W. Richard Stevens and Stephen A. Rago  
Addison-Wesley, 2005  
*We have probably referenced this book a few hundred thousand times. It is that useful to you, if you care to become an awesome systems programmer.*

## Homework

In this homework, we'll just familiarize ourselves with how the APIs described in the chapter work. To do so, you'll just write a few different programs, mostly based on various UNIX utilities.

### Questions

1. **Stat:** Write your own version of the command line program `stat`, which simply calls the `stat()` system call on a given file or directory. Print out file size, number of blocks allocated, reference (link) count, and so forth. What is the link count of a directory, as the number of entries in the directory changes? Useful interfaces: `stat()`
2. **List Files:** Write a program that lists files in the given directory. When called without any arguments, the program should just print the file names. When invoked with the `-l` flag, the program should print out information about each file, such as the owner, group, permissions, and other information obtained from the `stat()` system call. The program should take one additional argument, which is the directory to read, e.g., `myls -l directory`. If no directory is given, the program should just use the current working directory. Useful interfaces: `stat()`, `opendir()`, `readdir()`, `getcwd()`.
3. **Tail:** Write a program that prints out the last few lines of a file. The program should be efficient, in that it seeks to near the end of the file, reads in a block of data, and then goes backwards until it finds the requested number of lines; at this point, it should print out those lines from beginning to the end of the file. To invoke the program, one should type: `mytail -n file`, where `n` is the number of lines at the end of the file to print. Useful interfaces: `stat()`, `lseek()`, `open()`, `read()`, `close()`.
4. **Recursive Search:** Write a program that prints out the names of each file and directory in the file system tree, starting at a given point in the tree. For example, when run without arguments, the program should start with the current working directory and print its contents, as well as the contents of any sub-directories, etc., until the entire tree, root at the CWD, is printed. If given a single argument (of a directory name), use that as the root of the tree instead. Refine your recursive search with more fun options, similar to the powerful `find` command line tool. Useful interfaces: you figure it out.

## File System Implementation

In this chapter, we introduce a simple file system implementation, known as **vsfs** (the **Very Simple File System**). This file system is a simplified version of a typical UNIX file system and thus serves to introduce some of the basic on-disk structures, access methods, and various policies that you will find in many file systems today.

The file system is pure software; unlike our development of CPU and memory virtualization, we will not be adding hardware features to make some aspect of the file system work better (though we will want to pay attention to device characteristics to make sure the file system works well). Because of the great flexibility we have in building a file system, many different ones have been built, literally from AFS (the Andrew File System) [H+88] to ZFS (Sun's Zettabyte File System) [B07]. All of these file systems have different data structures and do some things better or worse than their peers. Thus, the way we will be learning about file systems is through case studies: first, a simple file system (vsfs) in this chapter to introduce most concepts, and then a series of studies of real file systems to understand how they can differ in practice.

### THE CRUX: HOW TO IMPLEMENT A SIMPLE FILE SYSTEM

How can we build a simple file system? What structures are needed on the disk? What do they need to track? How are they accessed?

### 40.1 The Way To Think

To think about file systems, we usually suggest thinking about two different aspects of them; if you understand both of these aspects, you probably understand how the file system basically works.

The first is the **data structures** of the file system. In other words, what types of on-disk structures are utilized by the file system to organize its data and metadata? The first file systems we'll see (including vsfs below) employ simple structures, like arrays of blocks or other objects, whereas

**ASIDE: MENTAL MODELS OF FILE SYSTEMS**

As we've discussed before, mental models are what you are really trying to develop when learning about systems. For file systems, your mental model should eventually include answers to questions like: what on-disk structures store the file system's data and metadata? What happens when a process opens a file? Which on-disk structures are accessed during a read or write? By working on and improving your mental model, you develop an abstract understanding of what is going on, instead of just trying to understand the specifics of some file-system code (though that is also useful, of course!).

more sophisticated file systems, like SGI's XFS, use more complicated tree-based structures [S+96].

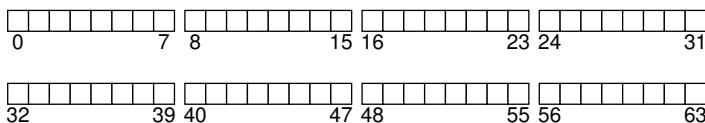
The second aspect of a file system is its **access methods**. How does it map the calls made by a process, such as `open()`, `read()`, `write()`, etc., onto its structures? Which structures are read during the execution of a particular system call? Which are written? How efficiently are all of these steps performed?

If you understand the data structures and access methods of a file system, you have developed a good mental model of how it truly works, a key part of the systems mindset. Try to work on developing your mental model as we delve into our first implementation.

## 40.2 Overall Organization

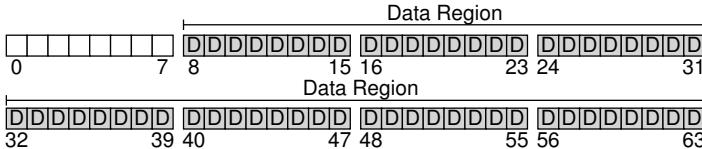
We now develop the overall on-disk organization of the data structures of the `vsfs` file system. The first thing we'll need to do is divide the disk into **blocks**; simple file systems use just one block size, and that's exactly what we'll do here. Let's choose a commonly-used size of 4 KB.

Thus, our view of the disk partition where we're building our file system is simple: a series of blocks, each of size 4 KB. The blocks are addressed from 0 to  $N - 1$ , in a partition of size  $N$  4-KB blocks. Assume we have a really small disk, with just 64 blocks:



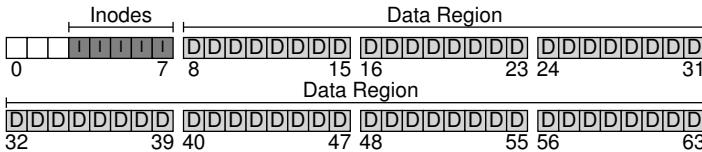
Let's now think about what we need to store in these blocks to build a file system. Of course, the first thing that comes to mind is user data. In fact, most of the space in any file system is (and should be) user data. Let's call the region of the disk we use for user data the **data region**, and,

again for simplicity, reserve a fixed portion of the disk for these blocks, say the last 56 of 64 blocks on the disk:



As we learned about (a little) last chapter, the file system has to track information about each file. This information is a key piece of **metadata**, and tracks things like which data blocks (in the data region) comprise a file, the size of the file, its owner and access rights, access and modify times, and other similar kinds of information. To store this information, file systems usually have a structure called an **inode** (we'll read more about inodes below).

To accommodate inodes, we'll need to reserve some space on the disk for them as well. Let's call this portion of the disk the **inode table**, which simply holds an array of on-disk inodes. Thus, our on-disk image now looks like this picture, assuming that we use 5 of our 64 blocks for inodes (denoted by I's in the diagram):

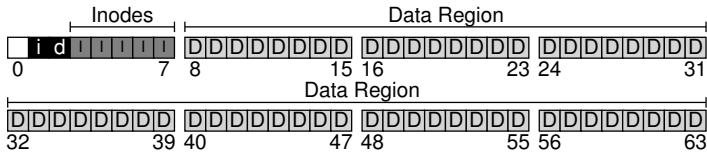


We should note here that inodes are typically not that big, for example 128 or 256 bytes. Assuming 256 bytes per inode, a 4-KB block can hold 16 inodes, and our file system above contains 80 total inodes. In our simple file system, built on a tiny 64-block partition, this number represents the maximum number of files we can have in our file system; however, do note that the same file system, built on a larger disk, could simply allocate a larger inode table and thus accommodate more files.

Our file system thus far has data blocks (D), and inodes (I), but a few things are still missing. One primary component that is still needed, as you might have guessed, is some way to track whether inodes or data blocks are free or allocated. Such **allocation structures** are thus a requisite element in any file system.

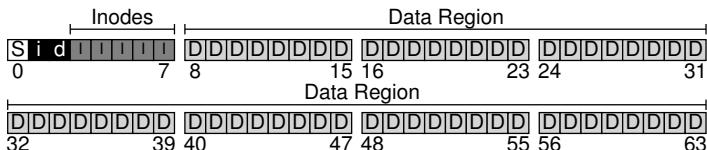
Many allocation-tracking methods are possible, of course. For example, we could use a **free list** that points to the first free block, which then points to the next free block, and so forth. We instead choose a simple and popular structure known as a **bitmap**, one for the data region (the **data bitmap**), and one for the inode table (the **inode bitmap**). A bitmap is a

simple structure: each bit is used to indicate whether the corresponding object/block is free (0) or in-use (1). And thus our new on-disk layout, with an inode bitmap (i) and a data bitmap (d):



You may notice that it is a bit of overkill to use an entire 4-KB block for these bitmaps; such a bitmap can track whether 32K objects are allocated, and yet we only have 80 inodes and 56 data blocks. However, we just use an entire 4-KB block for each of these bitmaps for simplicity.

The careful reader (i.e., the reader who is still awake) may have noticed there is one block left in the design of the on-disk structure of our very simple file system. We reserve this for the **superblock**, denoted by an S in the diagram below. The superblock contains information about this particular file system, including, for example, how many inodes and data blocks are in the file system (80 and 56, respectively in this instance), where the inode table begins (block 3), and so forth. It will likely also include a magic number of some kind to identify the file system type (in this case, vsfs).



Thus, when mounting a file system, the operating system will read the superblock first, to initialize various parameters, and then attach the volume to the file-system tree. When files within the volume are accessed, the system will thus know exactly where to look for the needed on-disk structures.

### 40.3 File Organization: The Inode

One of the most important on-disk structures of a file system is the **inode**; virtually all file systems have a structure similar to this. The name inode is short for **index node**, the historical name given to it in UNIX [RT74] and possibly earlier systems, used because these nodes were originally arranged in an array, and the array *indexed* into when accessing a particular inode.

### ASIDE: DATA STRUCTURE — THE INODE

The **inode** is the generic name that is used in many file systems to describe the structure that holds the metadata for a given file, such as its length, permissions, and the location of its constituent blocks. The name goes back at least as far as UNIX (and probably further back to Multics if not earlier systems); it is short for **index node**, as the inode number is used to index into an array of on-disk inodes in order to find the inode of that number. As we'll see, design of the inode is one key part of file system design. Most modern systems have some kind of structure like this for every file they track, but perhaps call them different things (such as dnodes, fnodes, etc.).

Each inode is implicitly referred to by a number (called the **inumber**), which we've earlier called the **low-level name** of the file. In vsfs (and other simple file systems), given an i-number, you should directly be able to calculate where on the disk the corresponding inode is located. For example, take the inode table of vsfs as above: 20-KB in size (5 4-KB blocks) and thus consisting of 80 inodes (assuming each inode is 256 bytes); further assume that the inode region starts at 12KB (i.e., the superblock starts at 0KB, the inode bitmap is at address 4KB, the data bitmap at 8KB, and thus the inode table comes right after). In vsfs, we thus have the following layout for the beginning of the file system partition (in closeup view):

			iblock 0	iblock 1	iblock 2	iblock 3	iblock 4	
Super	i-bmap	d-bmap	0 1 2 3 16 17 18 19 32 33 34 35 48 49 50 51 64 65 66 67	4 5 6 7 20 21 22 23 36 37 38 39 52 53 54 55 68 69 70 71	8 9 10 11 24 25 26 27 40 41 42 43 56 57 58 59 72 73 74 75	12 13 14 15 28 29 30 31 44 45 46 47 60 61 62 63 76 77 78 79		
0KB	4KB	8KB	12KB	16KB	20KB	24KB	28KB	32KB

To read inode number 32, the file system would first calculate the offset into the inode region ( $32 \cdot \text{sizeof(inode)} \text{ or } 8192$ ), add it to the start address of the inode table on disk ( $\text{inodeStartAddr} = 12KB$ ), and thus arrive upon the correct byte address of the desired block of inodes:  $20KB$ . Recall that disks are not byte addressable, but rather consist of a large number of addressable sectors, usually 512 bytes. Thus, to fetch the block of inodes that contains inode 32, the file system would issue a read to sector  $\frac{20 \times 1024}{512}$ , or 40, to fetch the desired inode block. More generally, the sector address  $\text{iaddr}$  of the inode block can be calculated as follows:

```
blk      = (inumber * sizeof(inode_t)) / blockSize;
sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;
```

Inside each inode is virtually all of the information you need about a file: its *type* (e.g., regular file, directory, etc.), its *size*, the number of *blocks*

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Figure 40.1: Simplified Ext2 Inode

allocated to it, *protection information* (such as who owns the file, as well as who can access it), some *time* information, including when the file was created, modified, or last accessed, as well as information about where its data blocks reside on disk (e.g., pointers of some kind). We refer to all such information about a file as **metadata**; in fact, any information inside the file system that isn't pure user data is often referred to as such. An example inode from ext2 [P09] is shown in Figure 40.1<sup>1</sup>.

One of the most important decisions in the design of the inode is how it refers to where data blocks are. One simple approach would be to have one or more **direct pointers** (disk addresses) inside the inode; each pointer refers to one disk block that belongs to the file. Such an approach is limited: for example, if you want to have a file that is really big (e.g., bigger than the size of a block multiplied by the number of direct pointers), you are out of luck.

## The Multi-Level Index

To support bigger files, file system designers have had to introduce different structures within inodes. One common idea is to have a special pointer known as an **indirect pointer**. Instead of pointing to a block that contains user data, it points to a block that contains more pointers, each of which point to user data. Thus, an inode may have some fixed number of direct pointers (e.g., 12), and a single indirect pointer. If a file grows large enough, an indirect block is allocated (from the data-block region of the disk), and the inode's slot for an indirect pointer is set to point to it. Assuming 4-KB blocks and 4-byte disk addresses, that adds another 1024 pointers; the file can grow to be  $(12 + 1024) \cdot 4K$  or 4144KB.

---

<sup>1</sup>Type info is kept in the directory entry, and thus is not found in the inode itself.

**TIP: CONSIDER EXTENT-BASED APPROACHES**

A different approach is to use **extents** instead of pointers. An extent is simply a disk pointer plus a length (in blocks); thus, instead of requiring a pointer for every block of a file, all one needs is a pointer and a length to specify the on-disk location of a file. Just a single extent is limiting, as one may have trouble finding a contiguous chunk of on-disk free space when allocating a file. Thus, extent-based file systems often allow for more than one extent, thus giving more freedom to the file system during file allocation.

In comparing the two approaches, pointer-based approaches are the most flexible but use a large amount of metadata per file (particularly for large files). Extent-based approaches are less flexible but more compact; in particular, they work well when there is enough free space on the disk and files can be laid out contiguously (which is the goal for virtually any file allocation policy anyhow).

Not surprisingly, in such an approach, you might want to support even larger files. To do so, just add another pointer to the inode: the **double indirect pointer**. This pointer refers to a block that contains pointers to indirect blocks, each of which contain pointers to data blocks. A double indirect block thus adds the possibility to grow files with an additional  $1024 \cdot 1024$  or 1-million 4KB blocks, in other words supporting files that are over 4GB in size. You may want even more, though, and we bet you know where this is headed: the **triple indirect pointer**.

Overall, this imbalanced tree is referred to as the **multi-level index** approach to pointing to file blocks. Let's examine an example with twelve direct pointers, as well as both a single and a double indirect block. Assuming a block size of 4 KB, and 4-byte pointers, this structure can accommodate a file of just over 4 GB in size (i.e.,  $(12 + 1024 + 1024^2) \times 4 \text{ KB}$ ). Can you figure out how big of a file can be handled with the addition of a triple-indirect block? (hint: pretty big)

Many file systems use a multi-level index, including commonly-used file systems such as Linux ext2 [P09] and ext3, NetApp's WAFL, as well as the original UNIX file system. Other file systems, including SGI XFS and Linux ext4, use **extents** instead of simple pointers; see the earlier aside for details on how extent-based schemes work (they are akin to segments in the discussion of virtual memory).

You might be wondering: why use an imbalanced tree like this? Why not a different approach? Well, as it turns out, many researchers have studied file systems and how they are used, and virtually every time they find certain “truths” that hold across the decades. One such finding is that *most files are small*. This imbalanced design reflects such a reality; if most files are indeed small, it makes sense to optimize for this case. Thus, with a small number of direct pointers (12 is a typical number), an inode

#### ASIDE: LINKED-BASED APPROACHES

Another simpler approach in designing inodes is to use a **linked list**. Thus, inside an inode, instead of having multiple pointers, you just need one, to point to the first block of the file. To handle larger files, add another pointer at the end of that data block, and so on, and thus you can support large files.

As you might have guessed, linked file allocation performs poorly for some workloads; think about reading the last block of a file, for example, or just doing random access. Thus, to make linked allocation work better, some systems will keep an in-memory table of link information, instead of storing the next pointers with the data blocks themselves. The table is indexed by the address of a data block  $D$ ; the content of an entry is simply  $D$ 's next pointer, i.e., the address of the next block in a file which follows  $D$ . A null-value could be there too (indicating an end-of-file), or some other marker to indicate that a particular block is free. Having such a table of next pointers makes it so that a linked allocation scheme can effectively do random file accesses, simply by first scanning through the (in memory) table to find the desired block, and then accessing (on disk) it directly.

Does such a table sound familiar? What we have described is the basic structure of what is known as the **file allocation table**, or **FAT** file system. Yes, this classic old Windows file system, before NTFS [C94], is based on a simple linked-based allocation scheme. There are other differences from a standard UNIX file system too; for example, there are no inodes per se, but rather directory entries which store metadata about a file and refer directly to the first block of said file, which makes creating hard links impossible. See Brouwer [B02] for more of the inelegant details.

can directly point to 48 KB of data, needing one (or more) indirect blocks for larger files. See Agrawal et. al [A+07] for a recent study; Figure 40.2 summarizes those results.

Of course, in the space of inode design, many other possibilities exist; after all, the inode is just a data structure, and any data structure that stores the relevant information, and can query it effectively, is sufficient. As file system software is readily changed, you should be willing to explore different designs should workloads or technologies change.

Most files are small	Roughly 2K is the most common size
Average file size is growing	Almost 200K is the average
Most bytes are stored in large files	A few big files use most of the space
File systems contains lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file systems remain ~50% full
Directories are typically small	Many have few entries; most have 20 or fewer

Figure 40.2: File System Measurement Summary

## 40.4 Directory Organization

In vsfs (as in many file systems), directories have a simple organization; a directory basically just contains a list of (entry name, inode number) pairs. For each file or directory in a given directory, there is a string and a number in the data block(s) of the directory. For each string, there may also be a length (assuming variable-sized names).

For example, assume a directory `dir` (inode number 5) has three files in it (`foo`, `bar`, and `foobar`), and their inode numbers are 12, 13, and 24 respectively. The on-disk data for `dir` might look like this:

inum	reclen	strlen	name
5	4	2	.
2	4	3	..
12	4	4	foo
13	4	4	bar
24	8	7	foobar

In this example, each entry has an inode number, record length (the total bytes for the name plus any left over space), string length (the actual length of the name), and finally the name of the entry. Note that each directory has two extra entries, `.` “dot” and `..` “dot-dot”; the dot directory is just the current directory (in this example, `dir`), whereas dot-dot is the parent directory (in this case, the root).

Deleting a file (e.g., calling `unlink()`) can leave an empty space in the middle of the directory, and hence there should be some way to mark that as well (e.g., with a reserved inode number such as zero). Such a delete is one reason the record length is used: a new entry may reuse an old, bigger entry and thus have extra space within.

You might be wondering where exactly directories are stored. Often, file systems treat directories as a special type of file. Thus, a directory has an inode, somewhere in the inode table (with the type field of the inode marked as “directory” instead of “regular file”). The directory has data blocks pointed to by the inode (and perhaps, indirect blocks); these data blocks live in the data block region of our simple file system. Our on-disk structure thus remains unchanged.

We should also note again that this simple linear list of directory entries is not the only way to store such information. As before, any data structure is possible. For example, XFS [S+96] stores directories in B-tree form, making file create operations (which have to ensure that a file name has not been used before creating it) faster than systems with simple lists that must be scanned in their entirety.

## 40.5 Free Space Management

A file system must track which inodes and data blocks are free, and which are not, so that when a new file or directory is allocated, it can find space for it. Thus **free space management** is important for all file systems. In vsfs, we have two simple bitmaps for this task.

**ASIDE: FREE SPACE MANAGEMENT**

There are many ways to manage free space; bitmaps are just one way. Some early file systems used **free lists**, where a single pointer in the super block was kept to point to the first free block; inside that block the next free pointer was kept, thus forming a list through the free blocks of the system. When a block was needed, the head block was used and the list updated accordingly.

Modern file systems use more sophisticated data structures. For example, SGI's XFS [S+96] uses some form of a **B-tree** to compactly represent which chunks of the disk are free. As with any data structure, different time-space trade-offs are possible.

For example, when we create a file, we will have to allocate an inode for that file. The file system will thus search through the bitmap for an inode that is free, and allocate it to the file; the file system will have to mark the inode as used (with a 1) and eventually update the on-disk bitmap with the correct information. A similar set of activities take place when a data block is allocated.

Some other considerations might also come into play when allocating data blocks for a new file. For example, some Linux file systems, such as ext2 and ext3, will look for a sequence of blocks (say 8) that are free when a new file is created and needs data blocks; by finding such a sequence of free blocks, and then allocating them to the newly-created file, the file system guarantees that a portion of the file will be contiguous on the disk, thus improving performance. Such a **pre-allocation** policy is thus a commonly-used heuristic when allocating space for data blocks.

## 40.6 Access Paths: Reading and Writing

Now that we have some idea of how files and directories are stored on disk, we should be able to follow the flow of operation during the activity of reading or writing a file. Understanding what happens on this **access path** is thus the second key in developing an understanding of how a file system works; pay attention!

For the following examples, let us assume that the file system has been mounted and thus that the superblock is already in memory. Everything else (i.e., inodes, directories) is still on the disk.

### Reading A File From Disk

In this simple example, let us first assume that you want to simply open a file (e.g., `/foo/bar`), read it, and then close it. For this simple example, let's assume the file is just 4KB in size (i.e., 1 block).

When you issue an `open ("/foo/bar", O_RDONLY)` call, the file system first needs to find the inode for the file `bar`, to obtain some basic information about the file (permissions information, file size, etc.). To do so,

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
			read			read				
open(bar)				read			read			
					read			read		
read()				read				read		
					write					
read()				read					read	
					write					
read()				read						read
					write					

Figure 40.3: File Read Timeline (Time Increasing Downward)

the file system must be able to find the inode, but all it has right now is the full pathname. The file system must **traverse** the pathname and thus locate the desired inode.

All traversals begin at the root of the file system, in the **root directory** which is simply called `/`. Thus, the first thing the FS will read from disk is the inode of the root directory. But where is this inode? To find an inode, we must know its i-number. Usually, we find the i-number of a file or directory in its parent directory; the root has no parent (by definition). Thus, the root inode number must be “well known”; the FS must know what it is when the file system is mounted. In most UNIX file systems, the root inode number is 2. Thus, to begin the process, the FS reads in the block that contains inode number 2 (the first inode block).

Once the inode is read in, the FS can look inside of it to find pointers to data blocks, which contain the contents of the root directory. The FS will thus use these on-disk pointers to read through the directory, in this case looking for an entry for `foo`. By reading in one or more directory data blocks, it will find the entry for `foo`; once found, the FS will also have found the inode number of `foo` (say it is 44) which it will need next.

The next step is to recursively traverse the pathname until the desired inode is found. In this example, the FS reads the block containing the inode of `foo` and then its directory data, finally finding the inode number of `bar`. The final step of `open()` is to read `bar`'s inode into memory; the FS then does a final permissions check, allocates a file descriptor for this process in the per-process open-file table, and returns it to the user.

Once `open`, the program can then issue a `read()` system call to read from the file. The first read (at offset 0 unless `lseek()` has been called) will thus read in the first block of the file, consulting the inode to find the location of such a block; it may also update the inode with a new last-accessed time. The read will further update the in-memory open file table for this file descriptor, updating the file offset such that the next read will read the second file block, etc.

**ASIDE: READS DON'T ACCESS ALLOCATION STRUCTURES**

We've seen many students get confused by allocation structures such as bitmaps. In particular, many often think that when you are simply reading a file, and not allocating any new blocks, that the bitmap will still be consulted. This is not true! Allocation structures, such as bitmaps, are only accessed when allocation is needed. The inodes, directories, and indirect blocks have all the information they need to complete a read request; there is no need to make sure a block is allocated when the inode already points to it.

At some point, the file will be closed. There is much less work to be done here; clearly, the file descriptor should be deallocated, but for now, that is all the FS really needs to do. No disk I/Os take place.

A depiction of this entire process is found in Figure 40.3 (time increases downward). In the figure, the open causes numerous reads to take place in order to finally locate the inode of the file. Afterwards, reading each block requires the file system to first consult the inode, then read the block, and then update the inode's last-accessed-time field with a write. Spend some time and try to understand what is going on.

Also note that the amount of I/O generated by the open is proportional to the length of the pathname. For each additional directory in the path, we have to read its inode as well as its data. Making this worse would be the presence of large directories; here, we only have to read one block to get the contents of a directory, whereas with a large directory, we might have to read many data blocks to find the desired entry. Yes, life can get pretty bad when reading a file; as you're about to find out, writing out a file (and especially, creating a new one) is even worse.

## Writing to Disk

Writing to a file is a similar process. First, the file must be opened (as above). Then, the application can issue `write()` calls to update the file with new contents. Finally, the file is closed.

Unlike reading, writing to the file may also **allocate** a block (unless the block is being overwritten, for example). When writing out a new file, each write not only has to write data to disk but has to first decide which block to allocate to the file and thus update other structures of the disk accordingly (e.g., the data bitmap and inode). Thus, each write to a file logically generates five I/Os: one to read the data bitmap (which is then updated to mark the newly-allocated block as used), one to write the bitmap (to reflect its new state to disk), two more to read and then write the inode (which is updated with the new block's location), and finally one to write the actual block itself.

The amount of write traffic is even worse when one considers a simple and common operation such as file creation. To create a file, the file system must not only allocate an inode, but also allocate space within

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
			read			read				
				read			read			
					read					
						write				
create (/foo/bar)		read write								
						read				
							read			
								write		
									write	
write()	read write									write
						write				
							read			
write()	read write									write
								write		
									write	
write()	read write									write

Figure 40.4: File Creation Timeline (Time Increasing Downward)

the directory containing the new file. The total amount of I/O traffic to do so is quite high: one read to the inode bitmap (to find a free inode), one write to the inode bitmap (to mark it allocated), one write to the new inode itself (to initialize it), one to the data of the directory (to link the high-level name of the file to its inode number), and one read and write to the directory inode to update it. If the directory needs to grow to accommodate the new entry, additional I/Os (i.e., to the data bitmap, and the new directory block) will be needed too. All that just to create a file!

Let's look at a specific example, where the file `/foo/bar` is created, and three blocks are written to it. Figure 40.4 shows what happens during the `open()` (which creates the file) and during each of three `4KB writes`.

In the figure, reads and writes to the disk are grouped under which system call caused them to occur, and the rough ordering they might take place in goes from top to bottom of the figure. You can see how much work it is to create the file: 10 I/Os in this case, to walk the pathname and then finally create the file. You can also see that each allocating write costs 5 I/Os: a pair to read and update the inode, another pair to read and update the data bitmap, and then finally the write of the data itself. How can a file system accomplish any of this with reasonable efficiency?

**THE CRUX: HOW TO REDUCE FILE SYSTEM I/O COSTS**

Even the simplest of operations like opening, reading, or writing a file incurs a huge number of I/O operations, scattered over the disk. What can a file system do to reduce the high costs of doing so many I/Os?

## 40.7 Caching and Buffering

As the examples above show, reading and writing files can be expensive, incurring many I/Os to the (slow) disk. To remedy what would clearly be a huge performance problem, most file systems aggressively use system memory (DRAM) to cache important blocks.

Imagine the open example above: without caching, every file open would require at least two reads for every level in the directory hierarchy (one to read the inode of the directory in question, and at least one to read its data). With a long pathname (e.g., `/1/2/3/ ... /100/file.txt`), the file system would literally perform hundreds of reads just to open the file!

Early file systems thus introduced a **fixed-size cache** to hold popular blocks. As in our discussion of virtual memory, strategies such as **LRU** and different variants would decide which blocks to keep in cache. This fixed-size cache would usually be allocated at boot time to be roughly 10% of total memory.

This **static partitioning** of memory, however, can be wasteful; what if the file system doesn't need 10% of memory at a given point in time? With the fixed-size approach described above, unused pages in the file cache cannot be re-purposed for some other use, and thus go to waste.

Modern systems, in contrast, employ a **dynamic partitioning** approach. Specifically, many modern operating systems integrate virtual memory pages and file system pages into a **unified page cache** [S00]. In this way, memory can be allocated more flexibly across virtual memory and file system, depending on which needs more memory at a given time.

Now imagine the file open example with caching. The first open may generate a lot of I/O traffic to read in directory inode and data, but subsequent file opens of that same file (or files in the same directory) will mostly hit in the cache and thus no I/O is needed.

Let us also consider the effect of caching on writes. Whereas read I/O can be avoided altogether with a sufficiently large cache, write traffic has to go to disk in order to become persistent. Thus, a cache does not serve as the same kind of filter on write traffic that it does for reads. That said, **write buffering** (as it is sometimes called) certainly has a number of performance benefits. First, by delaying writes, the file system can **batch** some updates into a smaller set of I/Os; for example, if an inode bitmap is updated when one file is created and then updated moments later as another file is created, the file system saves an I/O by delaying the write after the first update. Second, by buffering a number of writes in memory,

**TIP: UNDERSTAND STATIC VS. DYNAMIC PARTITIONING**

When dividing a resource among different clients/users, you can use either **static partitioning** or **dynamic partitioning**. The static approach simply divides the resource into fixed proportions once; for example, if there are two possible users of memory, you can give some fixed fraction of memory to one user, and the rest to the other. The dynamic approach is more flexible, giving out differing amounts of the resource over time; for example, one user may get a higher percentage of disk bandwidth for a period of time, but then later, the system may switch and decide to give a different user a larger fraction of available disk bandwidth.

Each approach has its advantages. Static partitioning ensures each user receives some share of the resource, usually delivers more predictable performance, and is often easier to implement. Dynamic partitioning can achieve better utilization (by letting resource-hungry users consume otherwise idle resources), but can be more complex to implement, and can lead to worse performance for users whose idle resources get consumed by others and then take a long time to reclaim when needed. As is often the case, there is no best method; rather, you should think about the problem at hand and decide which approach is most suitable. Indeed, shouldn't you always be doing that?

the system can then **schedule** the subsequent I/Os and thus increase performance. Finally, some writes are avoided altogether by delaying them; for example, if an application creates a file and then deletes it, delaying the writes to reflect the file creation to disk **avoids** them entirely. In this case, laziness (in writing blocks to disk) is a virtue.

For the reasons above, most modern file systems buffer writes in memory for anywhere between five and thirty seconds, representing yet another trade-off: if the system crashes before the updates have been propagated to disk, the updates are lost; however, by keeping writes in memory longer, performance can be improved by batching, scheduling, and even avoiding writes.

Some applications (such as databases) don't enjoy this trade-off. Thus, to avoid unexpected data loss due to write buffering, they simply force writes to disk, by calling `fsync()`, by using **direct I/O** interfaces that work around the cache, or by using the **raw disk** interface and avoiding the file system altogether<sup>2</sup>. While most applications live with the trade-offs made by the file system, there are enough controls in place to get the system to do what you want it to, should the default not be satisfying.

---

<sup>2</sup>Take a database class to learn more about old-school databases and their former insistence on avoiding the OS and controlling everything themselves. But watch out! Those database types are always trying to bad mouth the OS. Shame on you, database people. Shame.

**TIP: UNDERSTAND THE DURABILITY/PERFORMANCE TRADE-OFF**

Storage systems often present a durability/performance trade-off to users. If the user wishes data that is written to be immediately durable, the system must go through the full effort of committing the newly-written data to disk, and thus the write is slow (but safe). However, if the user can tolerate the loss of a little data, the system can buffer writes in memory for some time and write them later to the disk (in the background). Doing so makes writes appear to complete quickly, thus improving perceived performance; however, if a crash occurs, writes not yet committed to disk will be lost, and hence the trade-off. To understand how to make this trade-off properly, it is best to understand what the application using the storage system requires; for example, while it may be tolerable to lose the last few images downloaded by your web browser, losing part of a database transaction that is adding money to your bank account may be less tolerable. Unless you're rich, of course; in that case, why do you care so much about hoarding every last penny?

## 40.8 Summary

We have seen the basic machinery required in building a file system. There needs to be some information about each file (metadata), usually stored in a structure called an inode. Directories are just a specific type of file that store name→inode-number mappings. And other structures are needed too; for example, file systems often use a structure such as a bitmap to track which inodes or data blocks are free or allocated.

The terrific aspect of file system design is its freedom; the file systems we explore in the coming chapters each take advantage of this freedom to optimize some aspect of the file system. There are also clearly many policy decisions we have left unexplored. For example, when a new file is created, where should it be placed on disk? This policy and others will also be the subject of future chapters. Or will they?<sup>3</sup>

---

<sup>3</sup>Cue mysterious music that gets you even more intrigued about the topic of file systems.

## References

- [A+07] Nitin Agrawal, William J. Bolosky, John R. Douceur, Jacob R. Lorch  
 A Five-Year Study of File-System Metadata  
 FAST '07, pages 31–45, February 2007, San Jose, CA  
*An excellent recent analysis of how file systems are actually used. Use the bibliography within to follow the trail of file-system analysis papers back to the early 1980s.*
- [B07] “ZFS: The Last Word in File Systems”  
 Jeff Bonwick and Bill Moore  
 Available: <http://opensolaris.org/os/community/zfs/docs/zfs.last.pdf>  
*One of the most recent important file systems, full of features and awesomeness. We should have a chapter on it, and perhaps soon will.*
- [B02] “The FAT File System”  
 Andries Brouwer  
 September, 2002  
 Available: <http://www.win.tue.nl/~aeb/linux/fs/fat/fat.html>  
*A nice clean description of FAT. The file system kind, not the bacon kind. Though you have to admit, bacon fat probably tastes better.*
- [C94] “Inside the Windows NT File System”  
 Helen Custer  
 Microsoft Press, 1994  
*A short book about NTFS; there are probably ones with more technical details elsewhere.*
- [H+88] “Scale and Performance in a Distributed File System”  
 John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, Michael J. West.  
 ACM Transactions on Computing Systems (ACM TOCS), page 51-81, Volume 6, Number 1, February 1988  
*A classic distributed file system; we'll be learning more about it later, don't worry.*
- [P09] “The Second Extended File System: Internal Layout”  
 Dave Poirier, 2009  
 Available: <http://www.nongnu.org/ext2-doc/ext2.html>  
*Some details on ext2, a very simple Linux file system based on FFS, the Berkeley Fast File System. We'll be reading about it in the next chapter.*
- [RT74] “The UNIX Time-Sharing System”  
 M. Ritchie and K. Thompson  
 CACM, Volume 17:7, pages 365-375, 1974  
*The original paper about UNIX. Read it to see the underpinnings of much of modern operating systems.*
- [S00] “UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD”  
 Chuck Silvers  
 FREEUNIX, 2000  
*A nice paper about NetBSD's integration of file-system buffer caching and the virtual-memory page cache. Many other systems do the same type of thing.*
- [S+96] “Scalability in the XFS File System”  
 Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, Geoff Peck  
 USENIX '96, January 1996, San Diego, CA  
*The first attempt to make scalability of operations, including things like having millions of files in a directory, a central focus. A great example of pushing an idea to the extreme. The key idea behind this file system: everything is a tree. We should have a chapter on this file system too.*

## Homework

Use this tool, `vsfs.py`, to study how file system state changes as various operations take place. The file system begins in an empty state, with just a root directory. As the simulation takes place, various operations are performed, thus slowly changing the on-disk state of the file system. See the README for details.

## Questions

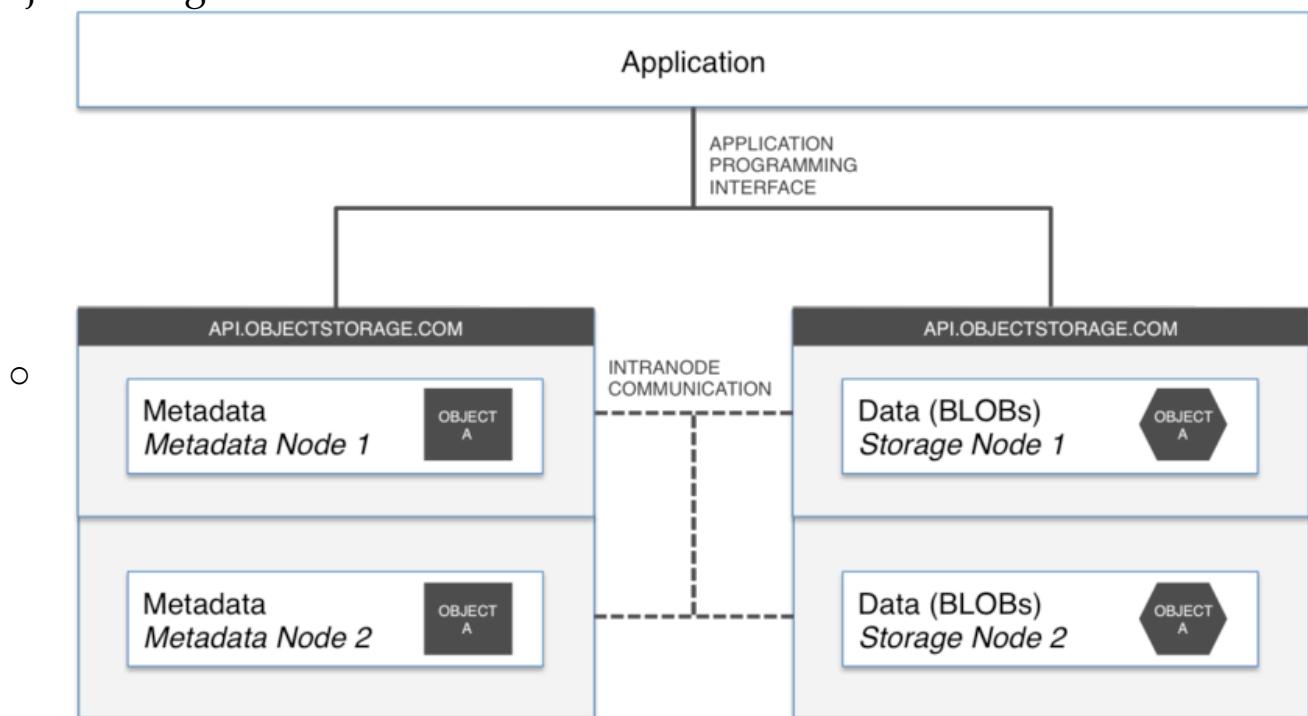
1. Run the simulator with some different random seeds (say 17, 18, 19, 20), and see if you can figure out which operations must have taken place between each state change.
2. Now do the same, using different random seeds (say 21, 22, 23, 24), except run with the `-r` flag, thus making you guess the state change while being shown the operation. What can you conclude about the inode and data-block allocation algorithms, in terms of which blocks they prefer to allocate?
3. Now reduce the number of data blocks in the file system, to very low numbers (say two), and run the simulator for a hundred or so requests. What types of files end up in the file system in this highly-constrained layout? What types of operations would fail?
4. Now do the same, but with inodes. With very few inodes, what types of operations can succeed? Which will usually fail? What is the final state of the file system likely to be?

# Wikipedia

Wednesday, November 9, 2016

10:23 PM

- File format
  - A file format is a standard way that information is encoded for storage in a computer file. It specifies how bits are used to encode information in a digital storage medium
- Object storage



- Storage architecture that manages data as objects that include the data itself, a variable amount of metadata and a globally unique identifier.
- Object storage can be implemented at multiple levels, including the device level (object storage device), the system level, and interface level.
- Abstraction of storage
  - Abstracts lower layers of storage away and data is exposed and managed as objects instead of files or blocks.
  - Do not have to deal with constructing or managing logical volumes to utilize disk capacity
  - Allows the addressing and identification of individual objects by more than just file name and file path
    - Adds a unique identifier within a bucket, or across the entire system, to support much larger namespaces and eliminate

name collisions.

- Inclusion of rich custom metadata within the object
  - PURPOSE
    - To capture application/user-specific information for better indexing purposes
    - To support data management policies
    - To centralize management of storage across many individual nodes and clusters
    - To optimize metadata storage independently from data storage
  - Data is organized into flexible-sized data containers called objects, rather than fixed sized blocks of data
  - Each object has both data and metadata
    - Physically encapsulating both together benefits recoverability
- Key-value database

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

- A data storage paradigm designed for storing, retrieving, and managing associative arrays known as a dictionary.
- Key-value systems treat the data as a single opaque collection which may have different fields for every record
  - + Offers considerable flexibility and more closely follows modern concepts like OOP
  - + Use far less memory to store the same database
  - \* Relational databases (RDB's) conversely structure the database with well defined data types

- Filesystem in Userspace (FUSE)

- Software interface that lets non-privileged users create their own file systems without editing kernel code
  - Achieved by running file system code in user space while the

FUSE model provides only a bridge to the actual kernel interfaces

- Often useful for writing virtual file systems

# An Introduction to DOS FAT Volume and File Structure

Mark Kampe [markk@cs.ucla.edu](mailto:markk@cs.ucla.edu)

## 1. Introduction

When the first personal computers with disks became available, they were very small (a few megabytes of disk and a few dozen kilobytes of memory). A file system implementation for such machines had to impose very little overhead on disk space, and be small enough to fit in the BIOS ROM. BIOS stands for **BASIC I/O Subsystem**. Note that the first word is all upper-case. The purpose of the BIOS ROM was to provide run-time support for a BASIC interpreter (which is what Bill Gates did for a living before building DOS). DOS was never intended to provide the features and performance of real timesharing systems.

Disk and memory size have increased in the last thirty years, People now demand state-of-the-art power and functionality from their PCs. Despite the evolution that the last decades have seen, old standards die hard. Much as European train tracks maintain the same wheel spacing used by Roman chariots, most modern OSs still support DOS FAT file systems. DOS file systems are not merely around for legacy reasons. The ISO 9660 CDROM file system format is a descendent of the DOS file system.

The DOS FAT file system is worth studying because:

- It is heavily used all over the world, and is the basis for more modern file system (like 9660).
- It provides reasonable performance (large transfers and well clustered allocation) with a very simple implementation.
- It is a very successful example of "linked list" space allocation.

## 2. Structural Overview

All file systems include a few basic types of data structures:

- bootstrap

code to be loaded into memory and executed when the computer is powered on. MVS volumes reserve the entire first track of the first cylinder for the boot strap.

- volume descriptors

information describing the size, type, and layout of the file system ... and in particular how to find the other key meta-data descriptors.

- file descriptors

information that describes a file (ownership, protection, time of last update, etc.) and points where the actual data is stored on the disk.

- free space descriptors

lists of blocks of (currently) unused space that can be allocated to files.

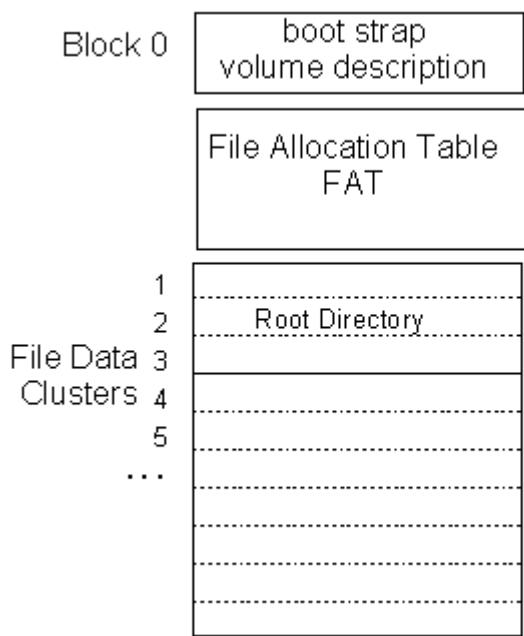
- file name descriptors

data structures that user-chosen names with each file.

DOS FAT file systems divide the volume into fixed-sized (physical) blocks, which are grouped into larger fixed-sized (logical) block clusters.

The first block of DOS FAT volume contains the bootstrap, along with some volume description information. After this comes a much longer **File Allocation Table** (FAT from which the file system takes its name). The File Allocation Table is used, both as a free list, and to keep track of which blocks have been allocated to which files.

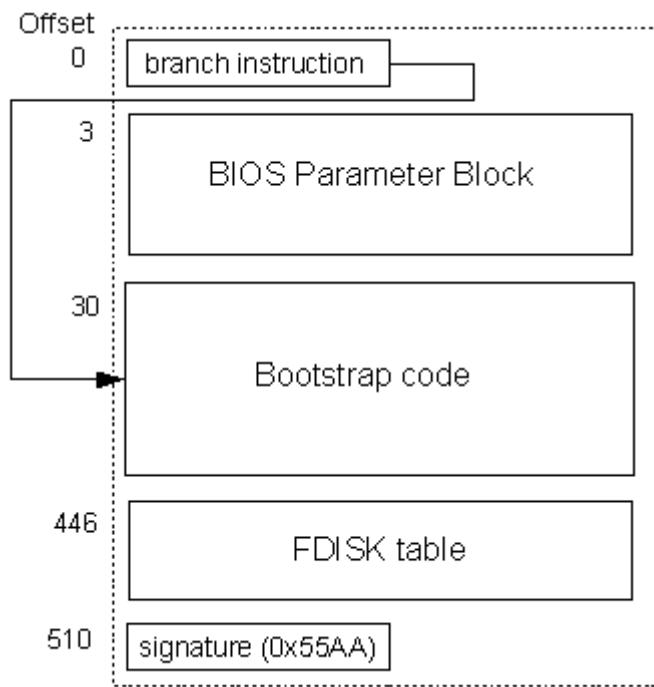
The remainder of the volume is data clusters, which can be allocated to files and directories. The first file on the volume is the root directory, the top of the tree from which all other files and directories on the volume can be reached.



### 3. Boot block BIOS Parameter Block and FDISK Table

Most file systems separate the first block (pure bootstrap code) from volume description information. DOS file systems often combine these into a single block. The format varies between (partitioned) hard disks and (unpartitioned) floppies, and between various releases of DOS and Windows ... but conceptually, the boot record:

- begins with a branch instruction (to the start of the real bootstrap code).
- followed by a volume description (BIOS Parameter Block)
- followed by the real bootstrap code
- followed by an optional disk partitioning table
- followed by a signature (for error checking).



### 3.1 BIOS Parameter Block

After the first few bytes of the bootstrap comes the BIOS parameter block, which contains a brief summary of the device and file system. It describes the device geometry:

- number of bytes per (physical) sector
- number of sectors per track
- number of tracks per cylinder
- total number of sectors on the volume

It also describes the way the file system is layed out on the volume:

- number of sectors per (logical) cluster
- the number of reserved sectors (not part of file system)
- the number of Alternate File Allocation Tables
- the number of entries in the root directory

These parameters enable the OS to interpret the remainder of the file system.

### 3.2 FDISK Table

As disks got larger, the people at MicroSoft figured out that their customers might want to put multiple file systems on each disk. This meant they needed some way of partitioning the disk into logical sub-disks. To do this, they added a small partition table (sometimes called the FDISK table, because of the program that managed it) to the end of the boot strap block.

This FDISK table has four entries, each capable of describing one disk partition. Each entry includes

- A partition type (e.g. Primary DOS partition, UNIX partition).
- An ACTIVE indication (is this the one we boot from).
- The disk address where that partition starts and ends.
- The number of sectors contained within that partition.

Partn	Type	Active	Start (C:H:S)	End (C:H:S)	Start (logical)	Size (sectors)
-------	------	--------	---------------	-------------	-----------------	----------------

1	LINUX	True	1:0:0	199:7:49	400	79,600
2	Windows NT		200:0:0	349:7:49	80,000	60,000
3	FAT 32		350:0:0	399:7:49	140,000	20,000
4	NONE					

In older versions of DOS the starting/ending addresses were specified as cylinder/sector/head. As disks got larger, this became less practical, and they moved to logical block numbers.

The addition of disk partitioning also changed the structure of the boot record. The first sector of a disk contains the Master Boot Record (MBR) which includes the FDISK table, and a bootstrap that finds the active partition, and reads in its first sector (Partition Boot Record). Most people (essentially everyone but Bill Gates :-) make their MBR bootstrap ask what system you want to boot from, and boot the active one by default after a few seconds. This gives you the opportunity to choose which OS you want to boot. Microsoft makes this decision for you ... you want to boot Windows.

The structure of the Partition Boot Record is entirely operating system and file system specific ... but for DOS FAT file system partitions, it includes a BIOS Parameter block as described above.

#### 4. File Descriptors (directories)

In keeping with their desire for simplicity, DOS file systems combine both file description and file naming into a single file descriptor (directory entries). A DOS directory is a file (of a special type) that contains a series of fixed sized (32 byte) directory entries. Each entry describes a single file:

- an 11-byte name (8 characters of base name, plus a 3 character extension).
- a byte of attribute bits for the file, which include:
  - Is this a file, or a sub-directory.
  - Has this file changed since the last backup.
  - Is this file hidden.
  - Is this file read-only.
  - Is this a system file.
  - Does this entry describe a volume label.
- times and dates of creation and last modification, and date of last access.
- a pointer to the first logical block of the file. (This field is only 16 bits wide, and so when Microsoft introduced the FAT32 file system, they had to put the high order bits in a different part of the directory entry).
- the length (number of valid data bytes) in the file.

Name (8+3)	Attributes	Last Changed	First Cluster	Length
.	DIR	08/01/03 11:15:00	61	2,048
..	DIR	06/20/03 08:10:24	1	4,096
MARK	DIR	10/15/04 21:40:12	130	1,800
README.TXT	FILE	11/02/04 04:27:36	410	31,280

If the first character of a files name is a NULL (0x00) the directory entry is unused. The special character (0xE5) in the first character of a file name is used to indicate that a directory entry describes a deleted file. (See the section on Garbage collection below)

##### Note on times and dates:

DOS stores file modification times and dates as a pair of 16-bit numbers:

- 7 bits of year, 4 bits of month, 5 bits of day of month
- 5 bits of hour, 6 bits of minute, 5 bits of seconds (x2).

All file systems use dates relative to some epoch (time zero). For DOS, the epoch is midnight, New Year's Eve, January 1, 1980. A seven bit field for years means that the DOS calendar only runs til 2107. Hopefully, nobody will still be using DOS file systems by then :-)

## 5. Links and Free Space (File Allocation Table)

Many file systems have very compact (e.g. bitmap) free lists, but most of them use some per-file data structure to keep track of which blocks are allocated to which file. The DOS File Allocation Table is a relatively unique design. It contains one entry for each logical block in the volume. If a block is free, this is indicated by the FAT entry. If a block is allocated to a file, the FAT entry gives the logical block number of the **next** logical block in the file.

### 5.1 Cluster Size and Performance

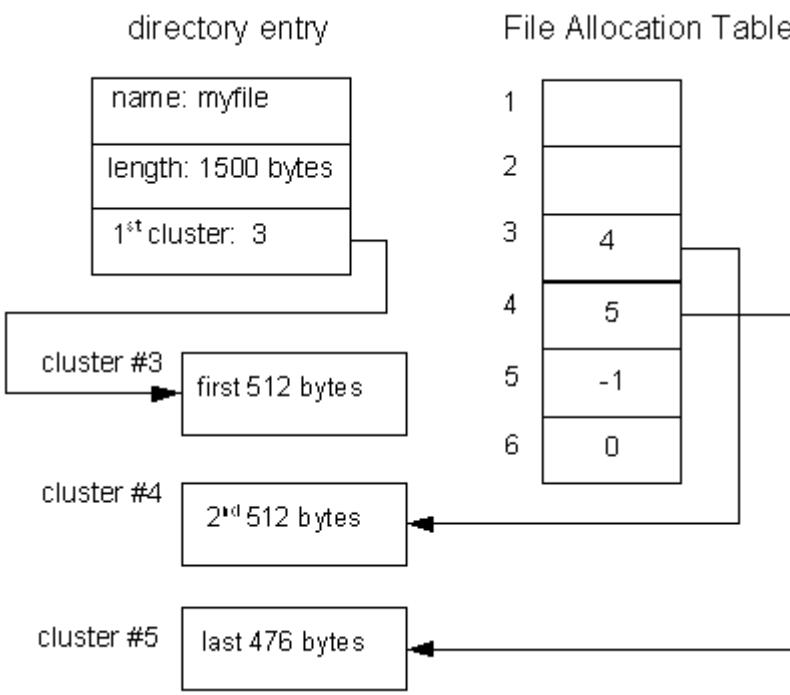
Space is allocated to files, not in (physical) blocks, but in (logical) multi-block clusters. The number of clusters per block is determined when the file system is created.

Allocating space to files in larger chunks improves I/O performance, by reducing the number of operations required to read or write a file. This comes at the cost of higher internal fragmentation (since, on average, half of the last cluster of each file is left unused). As disks have grown larger, people have become less concerned about internal fragmentation losses, and cluster sizes have increased.

The maximum number of clusters a volume can support depends on the width of the FAT entries. In the earliest FAT file systems (designed for use on floppies, and small hard drives). An 8-bit wide FAT entry would have been too small ( $256 * 512 = 128K$  bytes) to describe even the smallest floppy, but a 16-bit wide FAT entry would have been ludicrously large (8-16 Megabytes) ... so Microsoft compromised and adopted 12-bit wide FAT entries (two entries in three bytes). These were called FAT-12 file systems. As disks got larger, they created 2-byte wide (FAT-16) and 4-byte wide (FAT-32) file systems.

### 5.2 Next Block Pointers

A file's directory entry contains a pointer to the first cluster of that file. The File Allocation Table entry for that cluster tells us the cluster number **next** cluster in the file. When we finally get to the last cluster of the file, its FAT entry will contain a -1, indicating that there is no next block in the file.



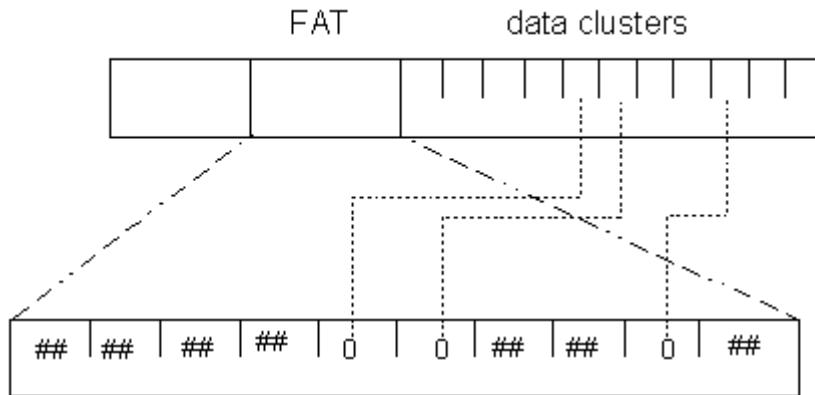
The "next block" organization of the FAT means that in order to figure out what physical cluster is the third logical block of a file, must know the physical cluster number of the second logical block. This is not usually a problem, because almost all file access is sequential (reading the first block, and then the second, and then the third ...).

If we had to go to disk to re-read the FAT each time we needed to figure out the next block number, the file system would perform very poorly. Fortunately, the FAT is so small (e.g. 512 bytes per megabyte of file system) that the entire FAT can be kept in memory as long as a file system is in use. This means that successor block numbers can be looked up without the need to do any additional disk I/O. It is easy to imagine

### 5.3 Free Space

The notion of "next block" is only meaningful for clusters that are allocated to a file ... which leaves us free to use the FAT entries associated with free clusters as a free indication. Just as we reserved a value (-1) to mean **end of file** we can reserve another value (0) to mean **this cluster is free**.

To find a free cluster, one has but to search the FAT for an entry with the value -2. If we want to find a free cluster near the clusters that are already allocated to the file, we can start our search with the FAT entry after the entry for the first cluster in the file.



Each FAT entry corresponds to one data cluster

A FAT entry of 0 means the corresponding data cluster is not allocated to any file.

## 5.4 Garbage Collection

Older versions of FAT file systems did not bother to free blocks when a file was deleted. Rather, they merely crossed out the first byte of the file name in the directory entry (with the reserved value 0xE5). This had the advantage of greatly reducing the amount of I/O associated with file deletion ... but it meant that DOS file systems regularly ran out of space.

When this happened, they would initiate garbage collection. Starting from the root directory, they would find every "valid" entry. They would follow the chain of next block pointers to determine which clusters were associated with each file, and recursively enumerate the contents of all sub-directories. After completing the enumeration of all allocated clusters, they inferred that any cluster not found in some file was free, and marked them as such in the File Allocation Table.

This "feature" was probably motivated by a combination of laziness and a desire for performance. It did, however, have an advantage. Since clusters were not freed when files were deleted, they could not be reallocated until after garbage collection was performed. This meant that it might be possible to recover the contents of deleted files for quite a while. The opportunity this created was large enough to enable Peter Norton to start a very successful company.

## 6. Descendents of the DOS file system

The DOS file system has evolved with time. Not only have wider (16- and 32-bit) FAT entries been used to support larger disks, but other features have been added. The last stand-alone DOS product was DOS 6.x. After this, all DOS support was under Windows, and along with the change to Windows came an enhanced version of the FAT file system called Virtual FAT (or simply VFAT).

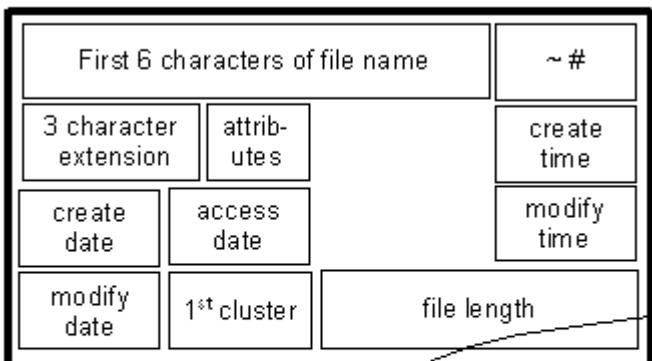
### 6.1 Long File Names

Most DOS and Windows systems were used for personal productivity, and their users didn't demand much in the way of file system features. Their biggest complaints were about the 8+3 file names. Windows users demanded longer and mixed-case file names.

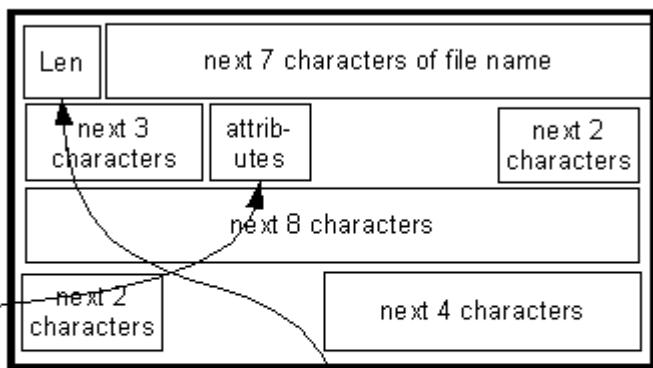
The 32 byte directory entries didn't have enough room to hold longer names, and changing the format of DOS directories would break hundreds or even thousands of applications. It wasn't merely a matter of importing files from old systems to new systems. DOS diskettes are commonly used to carry files between various systems ... which means that old systems still had to be able to read the new directories. They had to find a way to support longer file names without making the new files unreadable by older systems.

The solution they came up was to put the extended filenames in additional (auxiliary) directory entries. Each file would be described by an old-format directory entry, but supplementary directory entries (following the primary directory entry) could provide extensions to the file name. To keep older systems from being confused by the new directory entries, they were tagged with a very unusual set of attributes (hidden, system, read-only, volume label). Older systems would ignore new entries, but would still be able to access new files under their 8+3 names in the old-style directory entries. New systems would recognize the new directory entries, and be able to access files by either the long or the short names.

Primary (old style) directory entry



Secondary (continuation) directory entry



Attributes (Read Only, Hidden, System, Volume) identify this as an continuation directory entry. Older systems will ignore such entries.

Length field says how many more bytes of name are contained in this entry.

The addition of long file names did create one problem for the old directory entries. What would you do if you had two file names that differed only after the 8th character (e.g. datafileread.c and datafilewrite.c)? If we just used the first 8 characters of the file name in the old directory entries (datafile), we would have two files with the same name, and this is illegal. For this reason, the the short file names are not merely the first eight characters of the long file names. Rather, the last two bytes of the short name were merely made unambiguous (e.g. "~1", "~2", etc).

## 6.2 Alternate/back-up FATs

The File Allocation Table is a very concise way of keeping track of all of the next-block pointers in the file system. If anything ever happened to the File Allocation Table, the results would be disastrous. The directory entries would tell us where the first blocks of all files were, but we would have no way of figuring out where the remainder of the data was.

Events that corrupt the File Allocation Table are extremely rare, but the consequences of such an incident are catastrophic. To protect users from such eventualities, MicroSoft added support for alternate FATs. Periodically, the primary FAT would be copied to one of the pre-reserved alternat FAT locations. Then if something bad happened to the primary FAT, it would still be possible to read most files (files created before the copy) by using the back-up FAT. This is an imperfect solution, as losing new files is bad ... but losing all files is worse.

## 6.3 ISO 9660

When CDs were proposed for digital storage, everyone recognized the importance of a single standard file system format. Dueling formats would raise the cost of producing new products ... and this would be a lose for everyone. To respond to this need, the International Standards Organization chartered a sub-committee to propose such a standard.

The failings of the DOS file system were widely known by this time, but, as the most widely used file system on the planet, the committee members could not ignore it. Upon examination, it became clear that the most ideomatic features of the DOS file system (the File Allocation Table) were irrelevant to a CDROM file system (which is written only once):

- We don't need to keep track of the free space on a CD ROM. We write each file contiguously, and the next file goes immediately after the last one.
- Because files can be written contiguously, we don't need any "next block" pointers. All we need to know about a file is where its first block resides.

It was decided that ISO 9660 file systems would (like DOS file systems) have tree structured directory hierarchies, and that (like DOS) each directory entry would describe a single file (rather than having some auxiliary data structure like and I-node to do this). 9660 directory entries, like DOS directory entries, contain:

- file name (within the current directory)
- file type (e.g. file or directory)
- location of the file's first block
- number of bytes contained in the file
- time and date of creation

They did, however, learn from DOS's mistakes:

- Realizing that new information would be added to directory entries over time, they made them variable length. Each directory entry begins with a length field (giving the number of bytes in this directory entry, and thus the number of bytes until the next directory entry).
- Recognizing the need to support long file names, they also made the file name field in each entry a variable length field.
- Recognizing that, over time, people would want to associate a wide range of attributes with files, they also created a variable length extended attributes section after the file name. Much of this section has been left unused, but they defined several new attributes for files:
  - file owner
  - owning group
  - permissions
  - creation, modification, effective, and expiration times
  - record format, attributes, and length information

But, even though 9660 directory entries include much more information than DOS directory entries, it remains that 9660 volumes resemble DOS file systems much more than they resemble any other file system format. And so, the humble DOS file system is reborn in a new generation of products.

## 7. Summary

DOS file systems are very simple, They don't support multiple links to a file, or symbolic links, or even multi-user access control. They are also and very economical in terms of the space they take up. Free block lists, and file block pointers are combined into a single (quite compact) File Allocation Table. File descriptors are incorporated into the directory entries. And for all of these limitations, they are probably the most widely used

file system format on the planet. Despite their primitiveness, DOS file systems were used as the basis for much newer CD ROM file system designs.

What can we infer from this? That most users don't need a lot of fancy features, and that the DOS file system (primitive as it may be) covers their needs pretty well.

It is also noteworthy that when Microsoft was finally forced to change the file system format to get past the 8.3 upper case file name limitations, they chose to do so with a klugy (but upwards compatible) solution using additional directory entries per file. The fact that they chose such an implementation clearly illustrates the importance of maintaining media interchangability with older systems. This too is a problem that all (successful) file systems will eventually face.

## 8. References

DOS file system information

- PC Guide's [Overview of DOS FAT file systems](#). (this is a pointer to the long filename article ... but the entire library is nothing short of excellent).
- Free BSD sources, PCFS implementation, [BIOS Parameter block format](#), and (Open Solaris) [FDISK table format](#).
- Free BSD sources, PCFS implementation, [Directory Entry format](#)

9660 file system information

- Wikipedia Introduction to [ISO 9660 file systems](#)
- Free BSD sources, ISOFS implementation, [ISO 9660 data structures](#).

## Locality and The Fast File System

When the UNIX operating system was first introduced, the UNIX wizard himself Ken Thompson wrote the first file system. Let's call that the "old UNIX file system", and it was really simple. Basically, its data structures looked like this on the disk:



The super block (S) contained information about the entire file system: how big the volume is, how many inodes there are, a pointer to the head of a free list of blocks, and so forth. The inode region of the disk contained all the inodes for the file system. Finally, most of the disk was taken up by data blocks.

The good thing about the old file system was that it was simple, and supported the basic abstractions the file system was trying to deliver: files and the directory hierarchy. This easy-to-use system was a real step forward from the clumsy, record-based storage systems of the past, and the directory hierarchy was a true advance over simpler, one-level hierarchies provided by earlier systems.

### 41.1 The Problem: Poor Performance

The problem: performance was terrible. As measured by Kirk McKusick and his colleagues at Berkeley [MJLF84], performance started off bad and got worse over time, to the point where the file system was delivering only 2% of overall disk bandwidth!

The main issue was that the old UNIX file system treated the disk like it was a random-access memory; data was spread all over the place without regard to the fact that the medium holding the data was a disk, and thus had real and expensive positioning costs. For example, the data blocks of a file were often very far away from its inode, thus inducing an expensive seek whenever one first read the inode and then the data blocks of a file (a pretty common operation).

Worse, the file system would end up getting quite **fragmented**, as the free space was not carefully managed. The free list would end up pointing to a bunch of blocks spread across the disk, and as files got allocated, they would simply take the next free block. The result was that a logically contiguous file would be accessed by going back and forth across the disk, thus reducing performance dramatically.

For example, imagine the following data block region, which contains four files (A, B, C, and D), each of size 2 blocks:



If B and D are deleted, the resulting layout is:



As you can see, the free space is fragmented into two chunks of two blocks, instead of one nice contiguous chunk of four. Let's say you now wish to allocate a file E, of size four blocks:



You can see what happens: E gets spread across the disk, and as a result, when accessing E, you don't get peak (sequential) performance from the disk. Rather, you first read E1 and E2, then seek, then read E3 and E4. This fragmentation problem happened all the time in the old UNIX file system, and it hurt performance. A side note: this problem is exactly what disk **defragmentation** tools help with; they reorganize on-disk data to place files contiguously and make free space for one or a few contiguous regions, moving data around and then rewriting inodes and such to reflect the changes.

One other problem: the original block size was too small (512 bytes). Thus, transferring data from the disk was inherently inefficient. Smaller blocks were good because they minimized **internal fragmentation** (waste within the block), but bad for transfer as each block might require a positioning overhead to reach it. Thus, the problem:

#### THE CRUX:

#### HOW TO ORGANIZE ON-DISK DATA TO IMPROVE PERFORMANCE

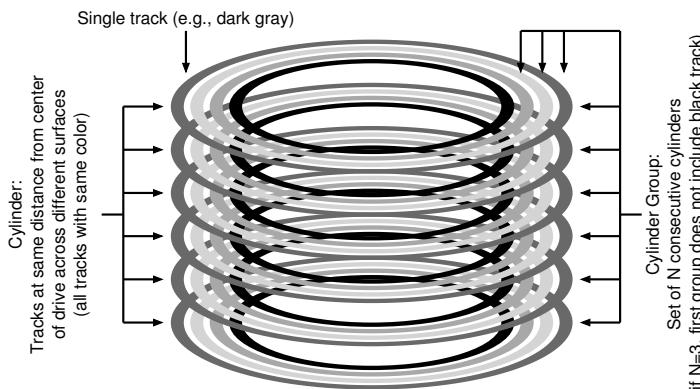
How can we organize file system data structures so as to improve performance? What types of allocation policies do we need on top of those data structures? How do we make the file system "disk aware"?

## 41.2 FFS: Disk Awareness Is The Solution

A group at Berkeley decided to build a better, faster file system, which they cleverly called the **Fast File System (FFS)**. The idea was to design the file system structures and allocation policies to be “disk aware” and thus improve performance, which is exactly what they did. FFS thus ushered in a new era of file system research; by keeping the same *interface* to the file system (the same APIs, including `open()`, `read()`, `write()`, `close()`, and other file system calls) but changing the internal *implementation*, the authors paved the path for new file system construction, work that continues today. Virtually all modern file systems adhere to the existing interface (and thus preserve compatibility with applications) while changing their internals for performance, reliability, or other reasons.

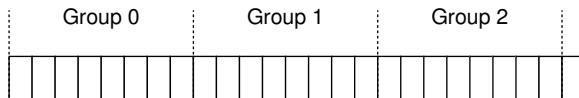
## 41.3 Organizing Structure: The Cylinder Group

The first step was to change the on-disk structures. FFS divides the disk into a number of **cylinder groups**. A single **cylinder** is a set of tracks on different surfaces of a hard drive that are the same distance from the center of the drive; it is called a cylinder because of its clear resemblance to the so-called geometrical shape. FFS aggregates each  $N$  consecutive cylinders into group, and thus the entire disk can thus be viewed as a collection of cylinder groups. Here is a simple example, showing the four outer most tracks of a drive with six platters, and a cylinder group that consists of three cylinders:



Note that modern drives do not export enough information for the file system to truly understand whether a particular cylinder is in use; as discussed previously [AD14a], disks export a logical address space of blocks and hide details of their geometry from clients. Thus, modern file

systems (such as Linux ext2, ext3, and ext4) instead organize the drive into **block groups**, each of which is just a consecutive portion of the disk's address space. The picture below illustrates an example where every 8 blocks are organized into a different block group (note that real groups would consist of many more blocks):



Whether you call them cylinder groups or block groups, these groups are the central mechanism that FFS uses to improve performance. Critically, by placing two files within the same group, FFS can ensure that accessing one after the other will not result in long seeks across the disk.

To use these groups to store files and directories, FFS needs to have the ability to place files and directories into a group, and track all necessary information about them therein. To do so, FFS includes all the structures you might expect a file system to have within each group, e.g., space for inodes, data blocks, and some structures to track whether each of those are allocated or free. Here is a depiction of what FFS keeps within a single cylinder group:



Let's now examine the components of this single cylinder group in more detail. FFS keeps a copy of the **super block** (S) in each group for reliability reasons. The super block is needed to mount the file system; by keeping multiple copies, if one copy becomes corrupt, you can still mount and access the file system by using a working replica.

Within each group, FFS needs to track whether the inodes and data blocks of the group are allocated. A per-group **inode bitmap** (ib) and **data bitmap** (db) serve this role for inodes and data blocks in each group. Bitmaps are an excellent way to manage free space in a file system because it is easy to find a large chunk of free space and allocate it to a file, perhaps avoiding some of the fragmentation problems of the free list in the old file system.

Finally, the **inode** and **data block** regions are just like those in the previous very-simple file system (VSFS). Most of each cylinder group, as usual, is comprised of data blocks.

**ASIDE: FFS FILE CREATION**

As an example, think about what data structures must be updated when a file is created; assume, for this example, that the user creates a new file `/foo/bar.txt` and that the file is one block long (4KB). The file is new, and thus needs a new inode; thus, both the inode bitmap and the newly-allocated inode will be written to disk. The file also has data in it and thus it too must be allocated; the data bitmap and a data block will thus (eventually) be written to disk. Hence, at least four writes to the current cylinder group will take place (recall that these writes may be buffered in memory for a while before they take place). But this is not all! In particular, when creating a new file, you must also place the file in the file-system hierarchy, i.e., the directory must be updated. Specifically, the parent directory `foo` must be updated to add the entry for `bar.txt`; this update may fit in an existing data block of `foo` or require a new block to be allocated (with associated data bitmap). The inode of `foo` must also be updated, both to reflect the new length of the directory as well as to update time fields (such as last-modified-time). Overall, it is a lot of work just to create a new file! Perhaps next time you do so, you should be more thankful, or at least surprised that it all works so well.

## 41.4 Policies: How To Allocate Files and Directories

With this group structure in place, FFS now has to decide how to place files and directories and associated metadata on disk to improve performance. The basic mantra is simple: *keep related stuff together* (and its corollary, *keep unrelated stuff far apart*).

Thus, to obey the mantra, FFS has to decide what is “related” and place it within the same block group; conversely, unrelated items should be placed into different block groups. To achieve this end, FFS makes use of a few simple placement heuristics.

The first is the placement of directories. FFS employs a simple approach: find the cylinder group with a low number of allocated directories (to balance directories across groups) and a high number of free inodes (to subsequently be able to allocate a bunch of files), and put the directory data and inode in that group. Of course, other heuristics could be used here (e.g., taking into account the number of free data blocks).

For files, FFS does two things. First, it makes sure (in the general case) to allocate the data blocks of a file in the same group as its inode, thus preventing long seeks between inode and data (as in the old file system). Second, it places all files that are in the same directory in the cylinder group of the directory they are in. Thus, if a user creates four files, `/a/b`, `/a/c`, `/a/d`, and `b/f`, FFS would try to place the first three near one another (same group) and the fourth far away (in some other group).

Let’s look at an example of such an allocation. In the example, assume that there are only 10 inodes and 10 data blocks in each group (both

unrealistically small numbers), and that the three directories (the root directory `/`, `/a`, and `/b`) and four files (`/a/c`, `/a/d`, `/a/e`, `/b/f`) are placed within them per the FFS policies. Assume the regular files are each two blocks in size, and that the directories have just a single block of data. For this figure, we use the obvious symbols for each file or directory (i.e., `/` for the root directory, `a` for `/a`, `f` for `/b/f`, and so forth).

```
group inodes      data
0 /----- /-----
1 acde----- accddee---
2 bf----- bff-----
3 ----- -----
4 ----- -----
5 -----
6 -----
7 ----- -----
...

```

Note that the FFS policy does two positive things: the data blocks of each file are near each file's inode, and files in the same directory are near one another (namely, `/a/c`, `/a/d`, and `/a/e` are all in Group 1, and directory `/b` and its file `/b/f` are near one another in Group 2).

In contrast, let's now look at an inode allocation policy that simply spreads inodes across groups, trying to ensure that no group's inode table fills up quickly. The final allocation might thus look something like this:

```
group inodes      data
0 /----- /-----
1 a----- a-----
2 b----- b-----
3 c----- cc-----
4 d----- dd-----
5 e----- ee-----
6 f----- ff-----
7 ----- -----
...

```

As you can see from the figure, while this policy does indeed keep file (and directory) data near its respective inode, files within a directory are arbitrarily spread around the disk, and thus name-based locality is not preserved. Access to files `/a/c`, `/a/d`, and `/a/e` now spans three groups instead of one as per the FFS approach.

The FFS policy heuristics are not based on extensive studies of file-system traffic or anything particularly nuanced; rather, they are based on good old-fashioned **common sense** (isn't that what CS stands for after all?)<sup>1</sup>. Files in a directory *are* often accessed together: imagine compiling a bunch of files and then linking them into a single executable. Because such namespace-based locality exists, FFS will often improve performance, making sure that seeks between related files are nice and short.

---

<sup>1</sup>Some people refer to common sense as **horse sense**, especially people who work regularly with horses. However, we have a feeling that this idiom may be lost as the "mechanized horse", a.k.a. the car, gains in popularity. What will they invent next? A flying machine??!

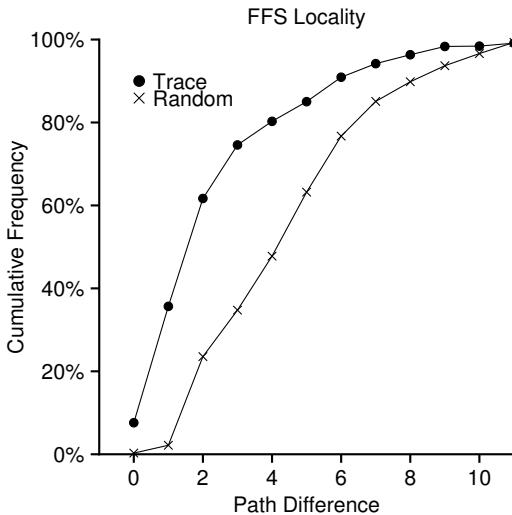


Figure 41.1: FFS Locality For SEER Traces

## 41.5 Measuring File Locality

To understand better whether these heuristics make sense, let's analyze some traces of file system access and see if indeed there is namespace locality. For some reason, there doesn't seem to be a good study of this topic in the literature.

Specifically, we'll use the SEER traces [K94] and analyze how "far away" file accesses were from one another in the directory tree. For example, if file *f* is opened, and then re-opened next in the trace (before any other files are opened), the distance between these two opens in the directory tree is zero (as they are the same file). If a file *f* in directory *dir* (i.e., *dir/f*) is opened, and followed by an open of file *g* in the same directory (i.e., *dir/g*), the distance between the two file accesses is one, as they share the same directory but are not the same file. Our distance metric, in other words, measures how far up the directory tree you have to travel to find the *common ancestor* of two files; the closer they are in the tree, the lower the metric.

Figure 41.1 shows the locality observed in the SEER traces over all workstations in the SEER cluster over the entirety of all traces. The graph plots the difference metric along the x-axis, and shows the cumulative percentage of file opens that were of that difference along the y-axis. Specifically, for the SEER traces (marked "Trace" in the graph), you can see that about 7% of file accesses were to the file that was opened previously, and that nearly 40% of file accesses were to either the same file or to one in the same directory (i.e., a difference of zero or one). Thus, the FFS locality assumption seems to make sense (at least for these traces).

Interestingly, another 25% or so of file accesses were to files that had a distance of two. This type of locality occurs when the user has structured a set of related directories in a multi-level fashion and consistently jumps between them. For example, if a user has a `src` directory and builds object files (`.o` files) into an `obj` directory, and both of these directories are sub-directories of a main `proj` directory, a common access pattern will be `proj/src/foo.c` followed by `proj/obj/foo.o`. The distance between these two accesses is two, as `proj` is the common ancestor. FFS does *not* capture this type of locality in its policies, and thus more seeking will occur between such accesses.

For comparison, the graph also shows locality for a “Random” trace. The random trace was generated by selecting files from within an existing SEER trace in random order, and calculating the distance metric between these randomly-ordered accesses. As you can see, there is less namespace locality in the random traces, as expected. However, because eventually every file shares a common ancestor (e.g., the root), there is some locality, and thus random is useful as a comparison point.

## 41.6 The Large-File Exception

In FFS, there is one important exception to the general policy of file placement, and it arises for large files. Without a different rule, a large file would entirely fill the block group it is first placed within (and maybe others). Filling a block group in this manner is undesirable, as it prevents subsequent “related” files from being placed within this block group, and thus may hurt file-access locality.

Thus, for large files, FFS does the following. After some number of blocks are allocated into the first block group (e.g., 12 blocks, or the number of direct pointers available within an inode), FFS places the next “large” chunk of the file (e.g., those pointed to by the first indirect block) in another block group (perhaps chosen for its low utilization). Then, the next chunk of the file is placed in yet another different block group, and so on.

Let’s look at some diagrams to understand this policy better. Without the large-file exception, a single large file would place all of its blocks into one part of the disk. We investigate a small example of a file (`/a`) with 30 blocks in an FFS configured with 10 inodes and 40 data blocks per group. Here is the depiction of FFS without the large-file exception:

group	inodes	data
0	/a-----	/aaaaaaaaaaaaaaa aaaaaaaaaaaa aaaaaaaaaaaa a-----
1	-----	-----
2	-----	-----
...		

As you can see in the picture, `/a` fills up most of the data blocks in Group 0, whereas other groups remain empty. If some other files are now created in the root directory (`/`), there is not much room for their data in the group.

With the large-file exception (here set to five blocks in each chunk), FFS instead spreads the file spread across groups, and the resulting utilization within any one group is not too high:

```
group inodes      data
 0 /a----- /aaaaaa-----
 1 ----- aaaaa----- -----
 2 ----- aaaaa----- -----
 3 ----- aaaaa----- -----
 4 ----- aaaaa----- -----
 5 ----- aaaaa----- -----
 6 ----- -----
 ...

```

The astute reader (that's you) will note that spreading blocks of a file across the disk will hurt performance, particularly in the relatively common case of sequential file access (e.g., when a user or application reads chunks 0 through 29 in order). And you are right, oh astute reader of ours! But you can address this problem by choosing chunk size carefully.

Specifically, if the chunk size is large enough, the file system will spend most of its time transferring data from disk and just a (relatively) little time seeking between chunks of the block. This process of reducing an overhead by doing more work per overhead paid is called **amortization** and is a common technique in computer systems.

Let's do an example: assume that the average positioning time (i.e., seek and rotation) for a disk is 10 ms. Assume further that the disk transfers data at 40 MB/s. If your goal was to spend half our time seeking between chunks and half our time transferring data (and thus achieve 50% of peak disk performance), you would thus need to spend 10 ms transferring data for every 10 ms positioning. So the question becomes: how big does a chunk have to be in order to spend 10 ms in transfer? Easy, just use our old friend, math, in particular the dimensional analysis mentioned in the chapter on disks [AD14a]:

$$\frac{40 \text{ MB}}{\text{sec}} \cdot \frac{1024 \text{ KB}}{1 \text{ MB}} \cdot \frac{1 \text{ sec}}{1000 \text{ ms}} \cdot 10 \text{ ms} = 409.6 \text{ KB} \quad (41.1)$$

Basically, what this equation says is this: if you transfer data at 40 MB/s, you need to transfer only 409.6KB every time you seek in order to spend half your time seeking and half your time transferring. Similarly, you can compute the size of the chunk you would need to achieve 90% of peak bandwidth (turns out it is about 3.69MB), or even 99% of peak bandwidth (40.6MB!). As you can see, the closer you want to get to peak, the bigger these chunks get (see Figure 41.2 for a plot of these values).

FFS did not use this type of calculation in order to spread large files across groups, however. Instead, it took a simple approach, based on the structure of the inode itself. The first twelve direct blocks were placed in the same group as the inode; each subsequent indirect block, and all the blocks it pointed to, was placed in a different group. With a block size of 4KB, and 32-bit disk addresses, this strategy implies that every

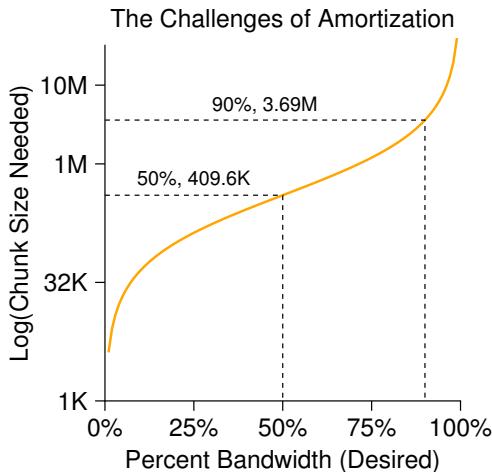


Figure 41.2: **Amortization: How Big Do Chunks Have To Be?**

1024 blocks of the file (4MB) were placed in separate groups, the lone exception being the first 48KB of the file as pointed to by direct pointers.

Note that the trend in disk drives is that transfer rate improves fairly rapidly, as disk manufacturers are good at cramming more bits into the same surface, but the mechanical aspects of drives related to seeks (disk arm speed and the rate of rotation) improve rather slowly [P98]. The implication is that over time, mechanical costs become relatively more expensive, and thus, to amortize said costs, you have to transfer more data between seeks.

## 41.7 A Few Other Things About FFS

FFS introduced a few other innovations too. In particular, the designers were extremely worried about accommodating small files; as it turned out, many files were 2KB or so in size back then, and using 4KB blocks, while good for transferring data, was not so good for space efficiency. This **internal fragmentation** could thus lead to roughly half the disk being wasted for a typical file system.

The solution the FFS designers hit upon was simple and solved the problem. They decided to introduce **sub-blocks**, which were 512-byte little blocks that the file system could allocate to files. Thus, if you created a small file (say 1KB in size), it would occupy two sub-blocks and thus not waste an entire 4KB block. As the file grew, the file system will continue allocating 512-byte blocks to it until it acquires a full 4KB of data. At that point, FFS will find a 4KB block, *copy* the sub-blocks into it, and free the sub-blocks for future use.

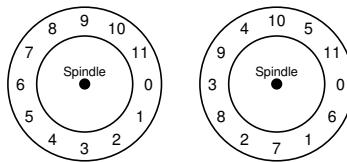


Figure 41.3: FFS: Standard Versus Parameterized Placement

You might observe that this process is inefficient, requiring a lot of extra work for the file system (in particular, a lot of extra I/O to perform the copy). And you'd be right again! Thus, FFS generally avoided this pesimial behavior by modifying the `libc` library; the library would buffer writes and then issue them in 4KB chunks to the file system, thus avoiding the sub-block specialization entirely in most cases.

A second neat thing that FFS introduced was a disk layout that was optimized for performance. In those times (before SCSI and other more modern device interfaces), disks were much less sophisticated and required the host CPU to control their operation in a more hands-on way. A problem arose in FFS when a file was placed on consecutive sectors of the disk, as on the left in Figure 41.3.

In particular, the problem arose during sequential reads. FFS would first issue a read to block 0; by the time the read was complete, and FFS issued a read to block 1, it was too late: block 1 had rotated under the head and now the read to block 1 would incur a full rotation.

FFS solved this problem with a different layout, as you can see on the right in Figure 41.3. By skipping over every other block (in the example), FFS has enough time to request the next block before it went past the disk head. In fact, FFS was smart enough to figure out for a particular disk *how many* blocks it should skip in doing layout in order to avoid the extra rotations; this technique was called **parameterization**, as FFS would figure out the specific performance parameters of the disk and use those to decide on the exact staggered layout scheme.

You might be thinking: this scheme isn't so great after all. In fact, you will only get 50% of peak bandwidth with this type of layout, because you have to go around each track twice just to read each block once. Fortunately, modern disks are much smarter: they internally read the entire track in and buffer it in an internal disk cache (often called a **track buffer** for this very reason). Then, on subsequent reads to the track, the disk will just return the desired data from its cache. File systems thus no longer have to worry about these incredibly low-level details. Abstraction and higher-level interfaces can be a good thing, when designed properly.

Some other usability improvements were added as well. FFS was one of the first file systems to allow for **long file names**, thus enabling more expressive names in the file system instead of the traditional fixed-size approach (e.g., 8 characters). Further, a new concept was introduced

**TIP: MAKE THE SYSTEM USABLE**

Probably the most basic lesson from FFS is that not only did it introduce the conceptually good idea of disk-aware layout, but it also added a number of features that simply made the system more usable. Long file names, symbolic links, and a rename operation that worked atomically all improved the utility of a system; while hard to write a research paper about (imagine trying to read a 14-pager about “The Symbolic Link: Hard Link’s Long Lost Cousin”), such small features made FFS more useful and thus likely increased its chances for adoption. Making a system usable is often as or more important than its deep technical innovations.

called a **symbolic link**. As discussed in a previous chapter [AD14b], hard links are limited in that they both could not point to directories (for fear of introducing loops in the file system hierarchy) and that they can only point to files within the same volume (i.e., the inode number must still be meaningful). Symbolic links allow the user to create an “alias” to any other file or directory on a system and thus are much more flexible. FFS also introduced an atomic `rename()` operation for renaming files. Usability improvements, beyond the basic technology, also likely gained FFS a stronger user base.

## 41.8 Summary

The introduction of FFS was a watershed moment in file system history, as it made clear that the problem of file management was one of the most interesting issues within an operating system, and showed how one might begin to deal with that most important of devices, the hard disk. Since that time, hundreds of new file systems have developed, but still today many file systems take cues from FFS (e.g., Linux ext2 and ext3 are obvious intellectual descendants). Certainly all modern systems account for the main lesson of FFS: treat the disk like it’s a disk.

## References

[AD14a] "Operating Systems: Three Easy Pieces"

*Chapter: Hard Disk Drives*

Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau

*There is no way you should be reading about FFS without having first understood hard drives in some detail. If you try to do so, please instead go directly to jail; do not pass go, and, critically, do not collect 200 much-needed simoleons.*

[AD14b] "Operating Systems: Three Easy Pieces"

*Chapter: File System Implementation*

Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau

*As above, it makes little sense to read this chapter unless you have read (and understood) the chapter on file system implementation. Otherwise, we'll be throwing around terms like "inode" and "indirect block" and you'll be like "huh?" and that is no fun for either of us.*

[K94] "The Design of the SEER Predictive Caching System"

G. H. Kuenning

MOBICOMM '94, Santa Cruz, California, December 1994

*According to Kuenning, this is the best overview of the SEER project, which led to (among other things) the collection of these traces.*

[MJLF84] "A Fast File System for UNIX"

Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry

ACM Transactions on Computing Systems, 2:3, pages 181-197.

*August, 1984. McKusick was recently honored with the IEEE Reynold B. Johnson award for his contributions to file systems, much of which was based on his work building FFS. In his acceptance speech, he discussed the original FFS software: only 1200 lines of code! Modern versions are a little more complex, e.g., the BSD FFS descendant now is in the 50-thousand lines-of-code range.*

[P98] "Hardware Technology Trends and Database Opportunities"

David A. Patterson

Keynote Lecture at the ACM SIGMOD Conference (SIGMOD '98)

June, 1998

*A great and simple overview of disk technology trends and how they change over time.*

## Homework

This section introduces `ffs.py`, a simple FFS simulator you can use to understand better how FFS-based file and directory allocation work. See the README for details on how to run the simulator.

### Questions

1. Examine the file `in.largefile`, and then run the simulator with flag `-f in.largefile` and `-L 4`. The latter sets the large file exception to 4 blocks in a group before moving on to the next one. What do you think the file system allocation will look like? Then run with `-c` enabled to see the actual layout.
2. Now run with `-L 30`. What do you expect to see? Once again, turn on `-c` to see if you were right. You can also use `-S` to see exactly which blocks were allocated to the file `/a`.
3. Now we will compute some statistics about the file. The first is something we call *filespan*, which is the max distance between any two data blocks of the file or between the inode and any data block. the file and the data block that is furthest away from it. Calculate the filespan of `/a`. Run `ffs.py -f in.largefile -L 4 -T -c` to see what it is. Do the same with `-L 100`. What difference do you expect in filespan as the large-file exception parameter changes from low values to high values?
4. Now let's look at a new input file, `in.manyfiles`. How do you think the FFS policy will lay these files out across groups? (you can run with `-v` to see what files and directories are created, or just `cat in.manyfiles`). Run the simulator with `-c` to see if you were right.
5. A new metric we will use to evaluate FFS is called *dirspan*. This metric calculates the spread of files within a particular directory, specifically the max distance between the inodes and data blocks of all the files in the directory as well as the inode and data block of the directory itself. Run with `in.manyfiles` and the `-T` flag, and see if you can figure out the dirspan of the three directories. Run with `-c` to see if you were right. How good of a job does FFS do in minimizing dirspan?
6. Now change the size of the inode table per group to 5 (`-I 5`). How do you think this will change the layout of the files? Run with `-c` to see if you were right. How does it affect the dirspan?
7. One policy that can affect FFS effectiveness is which group to place the inode of a new directory in. The default policy (in the simulator) simply looks for the group with the most free inodes. A slightly different policy, specified with `-A`, looks for a group of groups with the most free inodes. For example, if you run with `-A 2`, when allocating a new directory, the simulator will look at groups in pairs and pick the best pair for the allocation. Now you should run

- `./ffs.py -f in.manyfiles -I 5 -A 2 -c` to see how allocation changes with this strategy. How does it affect dirspan? Why might this policy be a good idea?
8. One last policy change we will explore relates to file fragmentation. Run `./ffs.py -f in.fragmented -v` and see if you can predict how the files that remain are allocated. Run with `-c` to confirm your answer. What is interesting about the data layout of file `/i`? Why is it problematic?
  9. A new policy, which we call *contiguous allocation* and enabled with the `-C` flag, tries to ensure that each file is allocated contiguously. Specifically, with `-C n`, the file system tries to ensure that `n` contiguous blocks are free within a group before allocating a block. Run `./ffs.py -f in.fragmented -v -C 2 -c` to see the difference in layout. How does layout change as the parameter passed to `-C` increases? Finally, how does `-C` affect filespan and dirspan?

## Crash Consistency: FSCK and Journaling

As we've seen thus far, the file system manages a set of data structures to implement the expected abstractions: files, directories, and all of the other metadata needed to support the basic abstraction that we expect from a file system. Unlike most data structures (for example, those found in memory of a running program), file system data structures must **persist**, i.e., they must survive over the long haul, stored on devices that retain data despite power loss (such as hard disks or flash-based SSDs).

One major challenge faced by a file system is how to update persistent data structures despite the presence of a **power loss** or **system crash**. Specifically, what happens if, right in the middle of updating on-disk structures, someone trips over the power cord and the machine loses power? Or the operating system encounters a bug and crashes? Because of power losses and crashes, updating a persistent data structure can be quite tricky, and leads to a new and interesting problem in file system implementation, known as the **crash-consistency problem**.

This problem is quite simple to understand. Imagine you have to update two on-disk structures, *A* and *B*, in order to complete a particular operation. Because the disk only services a single request at a time, one of these requests will reach the disk first (either *A* or *B*). If the system crashes or loses power after one write completes, the on-disk structure will be left in an **inconsistent** state. And thus, we have a problem that all file systems need to solve:

### THE CRUX: HOW TO UPDATE THE DISK DESPITE CRASHES

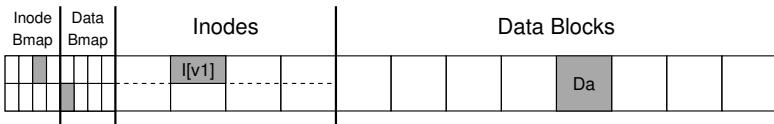
The system may crash or lose power between any two writes, and thus the on-disk state may only partially get updated. After the crash, the system boots and wishes to mount the file system again (in order to access files and such). Given that crashes can occur at arbitrary points in time, how do we ensure the file system keeps the on-disk image in a reasonable state?

In this chapter, we'll describe this problem in more detail, and look at some methods file systems have used to overcome it. We'll begin by examining the approach taken by older file systems, known as **fsck** or the **file system checker**. We'll then turn our attention to another approach, known as **journaling** (also known as **write-ahead logging**), a technique which adds a little bit of overhead to each write but recovers more quickly from crashes or power losses. We will discuss the basic machinery of journaling, including a few different flavors of journaling that Linux ext3 [T98,PAA05] (a relatively modern journaling file system) implements.

## 42.1 A Detailed Example

To kick off our investigation of journaling, let's look at an example. We'll need to use a **workload** that updates on-disk structures in some way. Assume here that the workload is simple: the append of a single data block to an existing file. The append is accomplished by opening the file, calling `lseek()` to move the file offset to the end of the file, and then issuing a single 4KB write to the file before closing it.

Let's also assume we are using standard simple file system structures on the disk, similar to file systems we have seen before. This tiny example includes an **inode bitmap** (with just 8 bits, one per inode), a **data bitmap** (also 8 bits, one per data block), inodes (8 total, numbered 0 to 7, and spread across four blocks), and data blocks (8 total, numbered 0 to 7). Here is a diagram of this file system:



If you look at the structures in the picture, you can see that a single inode is allocated (inode number 2), which is marked in the inode bitmap, and a single allocated data block (data block 4), also marked in the data bitmap. The inode is denoted  $I[v1]$ , as it is the first version of this inode; it will soon be updated (due to the workload described above).

Let's peek inside this simplified inode too. Inside of  $I[v1]$ , we see:

```
owner      : remzi
permissions : read-write
size       : 1
pointer    : 4
pointer    : null
pointer    : null
pointer    : null
```

In this simplified inode, the **size** of the file is 1 (it has one block allocated), the first direct pointer points to block 4 (the first data block of the file, Da), and all three other direct pointers are set to **null** (indicating

that they are not used). Of course, real inodes have many more fields; see previous chapters for more information.

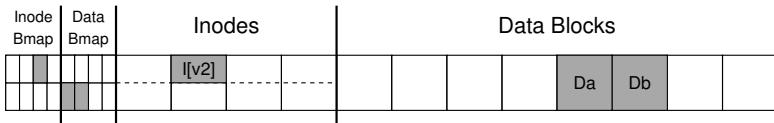
When we append to the file, we are adding a new data block to it, and thus must update three on-disk structures: the inode (which must point to the new block as well as have a bigger size due to the append), the new data block Db, and a new version of the data bitmap (call it B[v2]) to indicate that the new data block has been allocated.

Thus, in the memory of the system, we have three blocks which we must write to disk. The updated inode (inode version 2, or I[v2] for short) now looks like this:

```
owner      : remzi
permissions : read-write
size       : 2
pointer    : 4
pointer    : 5
pointer    : null
pointer    : null
```

The updated data bitmap (B[v2]) now looks like this: 00001100. Finally, there is the data block (Db), which is just filled with whatever it is users put into files. Stolen music perhaps?

What we would like is for the final on-disk image of the file system to look like this:



To achieve this transition, the file system must perform three separate writes to the disk, one each for the inode (I[v2]), bitmap (B[v2]), and data block (Db). Note that these writes usually don't happen immediately when the user issues a `write()` system call; rather, the dirty inode, bitmap, and new data will sit in main memory (in the **page cache** or **buffer cache**) for some time first; then, when the file system finally decides to write them to disk (after say 5 seconds or 30 seconds), the file system will issue the requisite write requests to the disk. Unfortunately, a crash may occur and thus interfere with these updates to the disk. In particular, if a crash happens after one or two of these writes have taken place, but not all three, the file system could be left in a funny state.

## Crash Scenarios

To understand the problem better, let's look at some example crash scenarios. Imagine only a single write succeeds; there are thus three possible outcomes, which we list here:

- **Just the data block (Db) is written to disk.** In this case, the data is on disk, but there is no inode that points to it and no bitmap that even says the block is allocated. Thus, it is as if the write never occurred. This case is not a problem at all, from the perspective of file-system crash consistency<sup>1</sup>.
- **Just the updated inode (I[v2]) is written to disk.** In this case, the inode points to the disk address (5) where Db was about to be written, but Db has not yet been written there. Thus, if we trust that pointer, we will read **garbage** data from the disk (the old contents of disk address 5).

Further, we have a new problem, which we call a **file-system inconsistency**. The on-disk bitmap is telling us that data block 5 has not been allocated, but the inode is saying that it has. This disagreement in the file system data structures is an inconsistency in the data structures of the file system; to use the file system, we must somehow resolve this problem (more on that below).

- **Just the updated bitmap (B[v2]) is written to disk.** In this case, the bitmap indicates that block 5 is allocated, but there is no inode that points to it. Thus the file system is inconsistent again; if left unresolved, this write would result in a **space leak**, as block 5 would never be used by the file system.

There are also three more crash scenarios in this attempt to write three blocks to disk. In these cases, two writes succeed and the last one fails:

- **The inode (I[v2]) and bitmap (B[v2]) are written to disk, but not data (Db).** In this case, the file system metadata is completely consistent: the inode has a pointer to block 5, the bitmap indicates that 5 is in use, and thus everything looks OK from the perspective of the file system's metadata. But there is one problem: 5 has garbage in it again.
- **The inode (I[v2]) and the data block (Db) are written, but not the bitmap (B[v2]).** In this case, we have the inode pointing to the correct data on disk, but again have an inconsistency between the inode and the old version of the bitmap (B1). Thus, we once again need to resolve the problem before using the file system.
- **The bitmap (B[v2]) and data block (Db) are written, but not the inode (I[v2]).** In this case, we again have an inconsistency between the inode and the data bitmap. However, even though the block was written and the bitmap indicates its usage, we have no idea which file it belongs to, as no inode points to the file.

---

<sup>1</sup>However, it might be a problem for the user, who just lost some data!

## The Crash Consistency Problem

Hopefully, from these crash scenarios, you can see the many problems that can occur to our on-disk file system image because of crashes: we can have inconsistency in file system data structures; we can have space leaks; we can return garbage data to a user; and so forth. What we'd like to do ideally is move the file system from one consistent state (e.g., before the file got appended to) to another **atomically** (e.g., after the inode, bitmap, and new data block have been written to disk). Unfortunately, we can't do this easily because the disk only commits one write at a time, and crashes or power loss may occur between these updates. We call this general problem the **crash-consistency problem** (we could also call it the **consistent-update problem**).

### 42.2 Solution #1: The File System Checker

Early file systems took a simple approach to crash consistency. Basically, they decided to let inconsistencies happen and then fix them later (when rebooting). A classic example of this lazy approach is found in a tool that does this: `fsck`<sup>2</sup>. `fsck` is a UNIX tool for finding such inconsistencies and repairing them [M86]; similar tools to check and repair a disk partition exist on different systems. Note that such an approach can't fix all problems; consider, for example, the case above where the file system looks consistent but the inode points to garbage data. The only real goal is to make sure the file system metadata is internally consistent.

The tool `fsck` operates in a number of phases, as summarized in McKusick and Kowalski's paper [MK96]. It is run *before* the file system is mounted and made available (`fsck` assumes that no other file-system activity is on-going while it runs); once finished, the on-disk file system should be consistent and thus can be made accessible to users.

Here is a basic summary of what `fsck` does:

- **Superblock:** `fsck` first checks if the superblock looks reasonable, mostly doing sanity checks such as making sure the file system size is greater than the number of blocks allocated. Usually the goal of these sanity checks is to find a suspect (corrupt) superblock; in this case, the system (or administrator) may decide to use an alternate copy of the superblock.
- **Free blocks:** Next, `fsck` scans the inodes, indirect blocks, double indirect blocks, etc., to build an understanding of which blocks are currently allocated within the file system. It uses this knowledge to produce a correct version of the allocation bitmaps; thus, if there is any inconsistency between bitmaps and inodes, it is resolved by trusting the information within the inodes. The same type of check is performed for all the inodes, making sure that all inodes that look like they are in use are marked as such in the inode bitmaps.

---

<sup>2</sup>Pronounced either "eff-ess-see-kay", "eff-ess-check", or, if you don't like the tool, "eff-suck". Yes, serious professional people use this term.

- **Inode state:** Each inode is checked for corruption or other problems. For example, `fsck` makes sure that each allocated inode has a valid type field (e.g., regular file, directory, symbolic link, etc.). If there are problems with the inode fields that are not easily fixed, the inode is considered suspect and cleared by `fsck`; the inode bitmap is correspondingly updated.
- **Inode links:** `fsck` also verifies the link count of each allocated inode. As you may recall, the link count indicates the number of different directories that contain a reference (i.e., a link) to this particular file. To verify the link count, `fsck` scans through the entire directory tree, starting at the root directory, and builds its own link counts for every file and directory in the file system. If there is a mismatch between the newly-calculated count and that found within an inode, corrective action must be taken, usually by fixing the count within the inode. If an allocated inode is discovered but no directory refers to it, it is moved to the `lost+found` directory.
- **Duplicates:** `fsck` also checks for duplicate pointers, i.e., cases where two different inodes refer to the same block. If one inode is obviously bad, it may be cleared. Alternately, the pointed-to block could be copied, thus giving each inode its own copy as desired.
- **Bad blocks:** A check for bad block pointers is also performed while scanning through the list of all pointers. A pointer is considered “bad” if it obviously points to something outside its valid range, e.g., it has an address that refers to a block greater than the partition size. In this case, `fsck` can’t do anything too intelligent; it just removes (clears) the pointer from the inode or indirect block.
- **Directory checks:** `fsck` does not understand the contents of user files; however, directories hold specifically formatted information created by the file system itself. Thus, `fsck` performs additional integrity checks on the contents of each directory, making sure that “.” and “..” are the first entries, that each inode referred to in a directory entry is allocated, and ensuring that no directory is linked to more than once in the entire hierarchy.

As you can see, building a working `fsck` requires intricate knowledge of the file system; making sure such a piece of code works correctly in all cases can be challenging [G+08]. However, `fsck` (and similar approaches) have a bigger and perhaps more fundamental problem: they are *too slow*. With a very large disk volume, scanning the entire disk to find all the allocated blocks and read the entire directory tree may take many minutes or hours. Performance of `fsck`, as disks grew in capacity and RAIDs grew in popularity, became prohibitive (despite recent advances [M+13]).

At a higher level, the basic premise of `fsck` seems just a tad irrational. Consider our example above, where just three blocks are written to the disk; it is incredibly expensive to scan the entire disk to fix problems that occurred during an update of just three blocks. This situation is akin to dropping your keys on the floor in your bedroom, and then com-

mencing a *search-the-entire-house-for-keys* recovery algorithm, starting in the basement and working your way through every room. It works but is wasteful. Thus, as disks (and RAIDs) grew, researchers and practitioners started to look for other solutions.

### 42.3 Solution #2: Journaling (or Write-Ahead Logging)

Probably the most popular solution to the consistent update problem is to steal an idea from the world of database management systems. That idea, known as **write-ahead logging**, was invented to address exactly this type of problem. In file systems, we usually call write-ahead logging **journaling** for historical reasons. The first file system to do this was Cedar [H87], though many modern file systems use the idea, including Linux ext3 and ext4, reiserfs, IBM's JFS, SGI's XFS, and Windows NTFS.

The basic idea is as follows. When updating the disk, before overwriting the structures in place, first write down a little note (somewhere else on the disk, in a well-known location) describing what you are about to do. Writing this note is the “write ahead” part, and we write it to a structure that we organize as a “log”; hence, write-ahead logging.

By writing the note to disk, you are guaranteeing that if a crash takes places during the update (overwrite) of the structures you are updating, you can go back and look at the note you made and try again; thus, you will know exactly what to fix (and how to fix it) after a crash, instead of having to scan the entire disk. By design, journaling thus adds a bit of work during updates to greatly reduce the amount of work required during recovery.

We'll now describe how **Linux ext3**, a popular journaling file system, incorporates journaling into the file system. Most of the on-disk structures are identical to **Linux ext2**, e.g., the disk is divided into block groups, and each block group has an inode and data bitmap as well as inodes and data blocks. The new key structure is the journal itself, which occupies some small amount of space within the partition or on another device. Thus, an ext2 file system (without journaling) looks like this:

Super	Group 0	Group 1	...	Group N	
-------	---------	---------	-----	---------	--

Assuming the journal is placed within the same file system image (though sometimes it is placed on a separate device, or as a file within the file system), an ext3 file system with a journal looks like this:

Super	Journal	Group 0	Group 1	...	Group N	
-------	---------	---------	---------	-----	---------	--

The real difference is just the presence of the journal, and of course, how it is used.

## Data Journaling

Let's look at a simple example to understand how **data journaling** works. Data journaling is available as a mode with the Linux ext3 file system, from which much of this discussion is based.

Say we have our canonical update again, where we wish to write the inode ( $I[v2]$ ), bitmap ( $B[v2]$ ), and data block ( $Db$ ) to disk again. Before writing them to their final disk locations, we are now first going to write them to the log (a.k.a. journal). This is what this will look like in the log:



You can see we have written five blocks here. The transaction begin ( $TxB$ ) tells us about this update, including information about the pending update to the file system (e.g., the final addresses of the blocks  $I[v2]$ ,  $B[v2]$ , and  $Db$ ), as well as some kind of **transaction identifier (TID)**. The middle three blocks just contain the exact contents of the blocks themselves; this is known as **physical logging** as we are putting the exact physical contents of the update in the journal (an alternate idea, **logical logging**, puts a more compact logical representation of the update in the journal, e.g., "this update wishes to append data block  $Db$  to file  $X$ ", which is a little more complex but can save space in the log and perhaps improve performance). The final block ( $TxE$ ) is a marker of the end of this transaction, and will also contain the TID.

Once this transaction is safely on disk, we are ready to overwrite the old structures in the file system; this process is called **checkpointing**. Thus, to **checkpoint** the file system (i.e., bring it up to date with the pending update in the journal), we issue the writes  $I[v2]$ ,  $B[v2]$ , and  $Db$  to their disk locations as seen above; if these writes complete successfully, we have successfully checkpointed the file system and are basically done. Thus, our initial sequence of operations:

1. **Journal write:** Write the transaction, including a transaction-begin block, all pending data and metadata updates, and a transaction-end block, to the log; wait for these writes to complete.
2. **Checkpoint:** Write the pending metadata and data updates to their final locations in the file system.

In our example, we would write  $TxB$ ,  $I[v2]$ ,  $B[v2]$ ,  $Db$ , and  $TxE$  to the journal first. When these writes complete, we would complete the update by checkpointing  $I[v2]$ ,  $B[v2]$ , and  $Db$ , to their final locations on disk.

Things get a little trickier when a crash occurs during the writes to the journal. Here, we are trying to write the set of blocks in the transaction (e.g.,  $TxB$ ,  $I[v2]$ ,  $B[v2]$ ,  $Db$ ,  $TxE$ ) to disk. One simple way to do this would be to issue each one at a time, waiting for each to complete, and then issuing the next. However, this is slow. Ideally, we'd like to issue

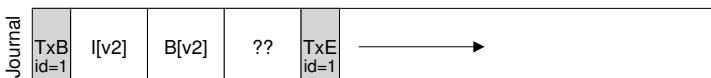
### ASIDE: FORCING WRITES TO DISK

To enforce ordering between two disk writes, modern file systems have to take a few extra precautions. In olden times, forcing ordering between two writes, *A* and *B*, was easy: just issue the write of *A* to the disk, wait for the disk to interrupt the OS when the write is complete, and then issue the write of *B*.

Things got slightly more complex due to the increased use of write caches within disks. With write buffering enabled (sometimes called **immediate reporting**), a disk will inform the OS the write is complete when it simply has been placed in the disk's memory cache, and has not yet reached disk. If the OS then issues a subsequent write, it is not guaranteed to reach the disk after previous writes; thus ordering between writes is not preserved. One solution is to disable write buffering. However, more modern systems take extra precautions and issue explicit **write barriers**; such a barrier, when it completes, guarantees that all writes issued before the barrier will reach disk before any writes issued after the barrier.

All of this machinery requires a great deal of trust in the correct operation of the disk. Unfortunately, recent research shows that some disk manufacturers, in an effort to deliver "higher performing" disks, explicitly ignore write-barrier requests, thus making the disks seemingly run faster but at the risk of incorrect operation [C+13, R+11]. As Kahan said, the fast almost always beats out the slow, even if the fast is wrong.

all five block writes at once, as this would turn five writes into a single sequential write and thus be faster. However, this is unsafe, for the following reason: given such a big write, the disk internally may perform scheduling and complete small pieces of the big write in any order. Thus, the disk internally may (1) write TxB, I[v2], B[v2], and TxE and only later (2) write Db. Unfortunately, if the disk loses power between (1) and (2), this is what ends up on disk:



Why is this a problem? Well, the transaction looks like a valid transaction (it has a begin and an end with matching sequence numbers). Further, the file system can't look at that fourth block and know it is wrong; after all, it is arbitrary user data. Thus, if the system now reboots and runs recovery, it will replay this transaction, and ignorantly copy the contents of the garbage block '??' to the location where Db is supposed to live. This is bad for arbitrary user data in a file; it is much worse if it happens to a critical piece of file system, such as the superblock, which could render the file system unmountable.

#### ASIDE: OPTIMIZING LOG WRITES

You may have noticed a particular inefficiency of writing to the log. Namely, the file system first has to write out the transaction-begin block and contents of the transaction; only after these writes complete can the file system send the transaction-end block to disk. The performance impact is clear, if you think about how a disk works: usually an extra rotation is incurred (think about why).

One of our former graduate students, Vijayan Prabhakaran, had a simple idea to fix this problem [P+05]. When writing a transaction to the journal, include a checksum of the contents of the journal in the begin and end blocks. Doing so enables the file system to write the entire transaction at once, without incurring a wait; if, during recovery, the file system sees a mismatch in the computed checksum versus the stored checksum in the transaction, it can conclude that a crash occurred during the write of the transaction and thus discard the file-system update. Thus, with a small tweak in the write protocol and recovery system, a file system can achieve faster common-case performance; on top of that, the system is slightly more reliable, as any reads from the journal are now protected by a checksum.

This simple fix was attractive enough to gain the notice of Linux file system developers, who then incorporated it into the next generation Linux file system, called (you guessed it!) **Linux ext4**. It now ships on millions of machines worldwide, including the Android handheld platform. Thus, every time you write to disk on many Linux-based systems, a little code developed at Wisconsin makes your system a little faster and more reliable.

To avoid this problem, the file system issues the transactional write in two steps. First, it writes all blocks except the TxE block to the journal, issuing these writes all at once. When these writes complete, the journal will look something like this (assuming our append workload again):

Journal	TxB id=1	I[v2]	B[v2]	Db	→
---------	-------------	-------	-------	----	---

When those writes complete, the file system issues the write of the TxE block, thus leaving the journal in this final, safe state:

Journal	TxB id=1	I[v2]	B[v2]	Db	TxE id=1	→
---------	-------------	-------	-------	----	-------------	---

An important aspect of this process is the atomicity guarantee provided by the disk. It turns out that the disk guarantees that any 512-byte

write will either happen or not (and never be half-written); thus, to make sure the write of TxE is atomic, one should make it a single 512-byte block. Thus, our current protocol to update the file system, with each of its three phases labeled:

1. **Journal write:** Write the contents of the transaction (including TxB, metadata, and data) to the log; wait for these writes to complete.
2. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for write to complete; transaction is said to be **committed**.
3. **Checkpoint:** Write the contents of the update (metadata and data) to their final on-disk locations.

## Recovery

Let's now understand how a file system can use the contents of the journal to **recover** from a crash. A crash may happen at any time during this sequence of updates. If the crash happens before the transaction is written safely to the log (i.e., before Step 2 above completes), then our job is easy: the pending update is simply skipped. If the crash happens after the transaction has committed to the log, but before the checkpoint is complete, the file system can **recover** the update as follows. When the system boots, the file system recovery process will scan the log and look for transactions that have committed to the disk; these transactions are thus **replayed** (in order), with the file system again attempting to write out the blocks in the transaction to their final on-disk locations. This form of logging is one of the simplest forms there is, and is called **redo logging**. By recovering the committed transactions in the journal, the file system ensures that the on-disk structures are consistent, and thus can proceed by mounting the file system and readying itself for new requests.

Note that it is fine for a crash to happen at any point during check-pointing, even after some of the updates to the final locations of the blocks have completed. In the worst case, some of these updates are simply performed again during recovery. Because recovery is a rare operation (only taking place after an unexpected system crash), a few redundant writes are nothing to worry about<sup>3</sup>.

## Batching Log Updates

You might have noticed that the basic protocol could add a lot of extra disk traffic. For example, imagine we create two files in a row, called `file1` and `file2`, in the same directory. To create one file, one has to update a number of on-disk structures, minimally including: the inode bitmap (to allocated a new inode), the newly-created inode of the file, the

---

<sup>3</sup>Unless you worry about everything, in which case we can't help you. Stop worrying so much, it is unhealthy! But now you're probably worried about over-worrying.

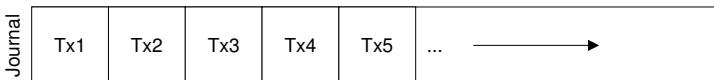
data block of the parent directory containing the new directory entry, as well as the parent directory inode (which now has a new modification time). With journaling, we logically commit all of this information to the journal for each of our two file creations; because the files are in the same directory, and assuming they even have inodes within the same inode block, this means that if we're not careful, we'll end up writing these same blocks over and over.

To remedy this problem, some file systems do not commit each update to disk one at a time (e.g., Linux ext3); rather, one can buffer all updates into a global transaction. In our example above, when the two files are created, the file system just marks the in-memory inode bitmap, inodes of the files, directory data, and directory inode as dirty, and adds them to the list of blocks that form the current transaction. When it is finally time to write these blocks to disk (say, after a timeout of 5 seconds), this single global transaction is committed containing all of the updates described above. Thus, by buffering updates, a file system can avoid excessive write traffic to disk in many cases.

## Making The Log Finite

We thus have arrived at a basic protocol for updating file-system on-disk structures. The file system buffers updates in memory for some time; when it is finally time to write to disk, the file system first carefully writes out the details of the transaction to the journal (a.k.a. write-ahead log); after the transaction is complete, the file system checkpoints those blocks to their final locations on disk.

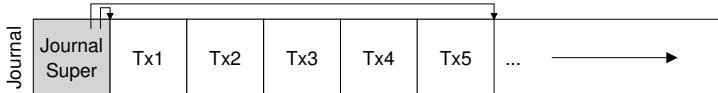
However, the log is of a finite size. If we keep adding transactions to it (as in this figure), it will soon fill. What do you think happens then?



Two problems arise when the log becomes full. The first is simpler, but less critical: the larger the log, the longer recovery will take, as the recovery process must replay all the transactions within the log (in order) to recover. The second is more of an issue: when the log is full (or nearly full), no further transactions can be committed to the disk, thus making the file system "less than useful" (i.e., useless).

To address these problems, journaling file systems treat the log as a circular data structure, re-using it over and over; this is why the journal is sometimes referred to as a **circular log**. To do so, the file system must take action some time after a checkpoint. Specifically, once a transaction has been checkpointed, the file system should free the space it was occupying within the journal, allowing the log space to be reused. There are many ways to achieve this end; for example, you could simply mark the

oldest and newest non-checkpointed transactions in the log in a **journal superblock**; all other space is free. Here is a graphical depiction:



In the journal superblock (not to be confused with the main file system superblock), the journaling system records enough information to know which transactions have not yet been checkpointed, and thus reduces recovery time as well as enables re-use of the log in a circular fashion. And thus we add another step to our basic protocol:

1. **Journal write:** Write the contents of the transaction (containing TxB and the contents of the update) to the log; wait for these writes to complete.
2. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction is now **committed**.
3. **Checkpoint:** Write the contents of the update to their final locations within the file system.
4. **Free:** Some time later, mark the transaction free in the journal by updating the journal superblock.

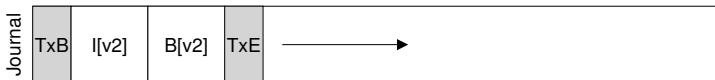
Thus we have our final data journaling protocol. But there is still a problem: we are writing each data block to the disk *twice*, which is a heavy cost to pay, especially for something as rare as a system crash. Can you figure out a way to retain consistency without writing data twice?

## Metadata Journaling

Although recovery is now fast (scanning the journal and replaying a few transactions as opposed to scanning the entire disk), normal operation of the file system is slower than we might desire. In particular, for each write to disk, we are now also writing to the journal first, thus doubling write traffic; this doubling is especially painful during sequential write workloads, which now will proceed at half the peak write bandwidth of the drive. Further, between writes to the journal and writes to the main file system, there is a costly seek, which adds noticeable overhead for some workloads.

Because of the high cost of writing every data block to disk twice, people have tried a few different things in order to speed up performance. For example, the mode of journaling we described above is often called **data journaling** (as in Linux ext3), as it journals all user data (in addition to the metadata of the file system). A simpler (and more common) form of journaling is sometimes called **ordered journaling** (or just **metadata**

**journaling**), and it is nearly the same, except that user data is *not* written to the journal. Thus, when performing the same update as above, the following information would be written to the journal:



The data block Db, previously written to the log, would instead be written to the file system proper, avoiding the extra write; given that most I/O traffic to the disk is data, not writing data twice substantially reduces the I/O load of journaling. The modification does raise an interesting question, though: when should we write data blocks to disk?

Let's again consider our example append of a file to understand the problem better. The update consists of three blocks: I[v2], B[v2], and Db. The first two are both metadata and will be logged and then checkpointed; the latter will only be written once to the file system. When should we write Db to disk? Does it matter?

As it turns out, the ordering of the data write does matter for metadata-only journaling. For example, what if we write Db to disk *after* the transaction (containing I[v2] and B[v2]) completes? Unfortunately, this approach has a problem: the file system is consistent but I[v2] may end up pointing to garbage data. Specifically, consider the case where I[v2] and B[v2] are written but Db did not make it to disk. The file system will then try to recover. Because Db is *not* in the log, the file system will replay writes to I[v2] and B[v2], and produce a consistent file system (from the perspective of file-system metadata). However, I[v2] will be pointing to garbage data, i.e., at whatever was in the slot where Db was headed.

To ensure this situation does not arise, some file systems (e.g., Linux ext3) write data blocks (of regular files) to the disk *first*, before related metadata is written to disk. Specifically, the protocol is as follows:

1. **Data write:** Write data to final location; wait for completion (the wait is optional; see below for details).
2. **Journal metadata write:** Write the begin block and metadata to the log; wait for writes to complete.
3. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction (including data) is now **committed**.
4. **Checkpoint metadata:** Write the contents of the metadata update to their final locations within the file system.
5. **Free:** Later, mark the transaction free in journal superblock.

By forcing the data write first, a file system can guarantee that a pointer will never point to garbage. Indeed, this rule of "write the pointed to object before the object with the pointer to it" is at the core of crash consistency, and is exploited even further by other crash consistency schemes [GP94] (see below for details).

In most systems, metadata journaling (akin to ordered journaling of ext3) is more popular than full data journaling. For example, Windows NTFS and SGI's XFS both use some form of metadata journaling. Linux ext3 gives you the option of choosing either data, ordered, or unordered modes (in unordered mode, data can be written at any time). All of these modes keep metadata consistent; they vary in their semantics for data.

Finally, note that forcing the data write to complete (Step 1) before issuing writes to the journal (Step 2) is not required for correctness, as indicated in the protocol above. Specifically, it would be fine to issue data writes as well as the transaction-begin block and metadata to the journal; the only real requirement is that Steps 1 and 2 complete before the issuing of the journal commit block (Step 3).

### Tricky Case: Block Reuse

There are some interesting corner cases that make journaling more tricky, and thus are worth discussing. A number of them revolve around block reuse; as Stephen Tweedie (one of the main forces behind ext3) said:

“What’s the hideous part of the entire system? ... It’s deleting files. Everything to do with delete is hairy. Everything to do with delete... you have nightmares around what happens if blocks get deleted and then reallocated.” [T00]

The particular example Tweedie gives is as follows. Suppose you are using some form of metadata journaling (and thus data blocks for files are *not* journaled). Let’s say you have a directory called `foo`. The user adds an entry to `foo` (say by creating a file), and thus the contents of `foo` (because directories are considered metadata) are written to the log; assume the location of the `foo` directory data is block 1000. The log thus contains something like this:

Journal	TxB id=1	I[foo] ptr:1000	D[foo] [final addr:1000]	TxE id=1	→
---------	-------------	--------------------	-----------------------------	-------------	---

At this point, the user deletes everything in the directory as well as the directory itself, freeing up block 1000 for reuse. Finally, the user creates a new file (say `foobar`), which ends up reusing the same block (1000) that used to belong to `foo`. The inode of `foobar` is committed to disk, as is its data; note, however, because metadata journaling is in use, only the inode of `foobar` is committed to the journal; the newly-written data in block 1000 in the file `foobar` is *not* journaled.

Journal	TxB id=1	I[foo] ptr:1000	D[foo] [final addr:1000]	TxE id=1	TxB id=2	I[foobar] ptr:1000	TxE id=2	→
---------	-------------	--------------------	-----------------------------	-------------	-------------	-----------------------	-------------	---

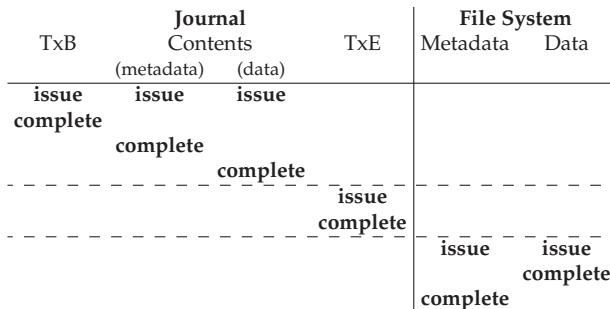


Figure 42.1: Data Journaling Timeline

Now assume a crash occurs and all of this information is still in the log. During replay, the recovery process simply replays everything in the log, including the write of directory data in block 1000; the replay thus overwrites the user data of current file `foobar` with old directory contents! Clearly this is not a correct recovery action, and certainly it will be a surprise to the user when reading the file `foobar`.

There are a number of solutions to this problem. One could, for example, never reuse blocks until the delete of said blocks is checkpointed out of the journal. What Linux ext3 does instead is to add a new type of record to the journal, known as a **revoke** record. In the case above, deleting the directory would cause a revoke record to be written to the journal. When replaying the journal, the system first scans for such revoke records; any such revoked data is never replayed, thus avoiding the problem mentioned above.

## Wrapping Up Journaling: A Timeline

Before ending our discussion of journaling, we summarize the protocols we have discussed with timelines depicting each of them. Figure 42.1 shows the protocol when journaling data as well as metadata, whereas Figure 42.2 shows the protocol when journaling only metadata.

In each figure, time increases in the downward direction, and each row in the figure shows the logical time that a write can be issued or might complete. For example, in the data journaling protocol (Figure 42.1), the writes of the transaction begin block (TxB) and the contents of the transaction can logically be issued at the same time, and thus can be completed in any order; however, the write to the transaction end block (TxE) must not be issued until said previous writes complete. Similarly, the checkpointing writes to data and metadata blocks cannot begin until the transaction end block has committed. Horizontal dashed lines show where write-ordering requirements must be obeyed.

A similar timeline is shown for the metadata journaling protocol. Note that the data write can logically be issued at the same time as the writes

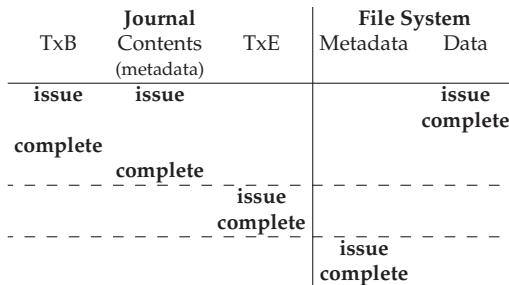


Figure 42.2: Metadata Journaling Timeline

to the transaction begin and the contents of the journal; however, it must be issued and complete before the transaction end has been issued.

Finally, note that the time of completion marked for each write in the timelines is arbitrary. In a real system, completion time is determined by the I/O subsystem, which may reorder writes to improve performance. The only guarantees about ordering that we have are those that must be enforced for protocol correctness (and are shown via the horizontal dashed lines in the figures).

## 42.4 Solution #3: Other Approaches

We've thus far described two options in keeping file system metadata consistent: a lazy approach based on `fsck`, and a more active approach known as journaling. However, these are not the only two approaches. One such approach, known as Soft Updates [GP94], was introduced by Ganger and Patt. This approach carefully orders all writes to the file system to ensure that the on-disk structures are never left in an inconsistent state. For example, by writing a pointed-to data block to disk *before* the inode that points to it, we can ensure that the inode never points to garbage; similar rules can be derived for all the structures of the file system. Implementing Soft Updates can be a challenge, however; whereas the journaling layer described above can be implemented with relatively little knowledge of the exact file system structures, Soft Updates requires intricate knowledge of each file system data structure and thus adds a fair amount of complexity to the system.

Another approach is known as **copy-on-write** (yes, COW), and is used in a number of popular file systems, including Sun's ZFS [B07]. This technique never overwrites files or directories in place; rather, it places new updates to previously unused locations on disk. After a number of updates are completed, COW file systems flip the root structure of the file system to include pointers to the newly updated structures. Doing so makes keeping the file system consistent straightforward. We'll be learning more about this technique when we discuss the log-structured file system (LFS) in a future chapter; LFS is an early example of a COW.

Another approach is one we just developed here at Wisconsin. In this technique, entitled **backpointer-based consistency** (or **BBC**), no ordering is enforced between writes. To achieve consistency, an additional **back pointer** is added to every block in the system; for example, each data block has a reference to the inode to which it belongs. When accessing a file, the file system can determine if the file is consistent by checking if the forward pointer (e.g., the address in the inode or direct block) points to a block that refers back to it. If so, everything must have safely reached disk and thus the file is consistent; if not, the file is inconsistent, and an error is returned. By adding back pointers to the file system, a new form of lazy crash consistency can be attained [C+12].

Finally, we also have explored techniques to reduce the number of times a journal protocol has to wait for disk writes to complete. Entitled **optimistic crash consistency** [C+13], this new approach issues as many writes to disk as possible and uses a generalized form of the **transaction checksum** [P+05], as well as a few other techniques, to detect inconsistencies should they arise. For some workloads, these optimistic techniques can improve performance by an order of magnitude. However, to truly function well, a slightly different disk interface is required [C+13].

## 42.5 Summary

We have introduced the problem of crash consistency, and discussed various approaches to attacking this problem. The older approach of building a file system checker works but is likely too slow to recover on modern systems. Thus, many file systems now use journaling. Journaling reduces recovery time from  $O(\text{size-of-the-disk-volume})$  to  $O(\text{size-of-the-log})$ , thus speeding recovery substantially after a crash and restart. For this reason, many modern file systems use journaling. We have also seen that journaling can come in many different forms; the most commonly used is ordered metadata journaling, which reduces the amount of traffic to the journal while still preserving reasonable consistency guarantees for both file system metadata as well as user data.

## References

[B07] "ZFS: The Last Word in File Systems"

Jeff Bonwick and Bill Moore

Available: <http://opensolaris.org/os/community/zfs/docs/zfs.last.pdf>

*ZFS uses copy-on-write and journaling, actually, as in some cases, logging writes to disk will perform better.*

[C+12] "Consistency Without Ordering"

Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau  
FAST '12, San Jose, California

*A recent paper of ours about a new form of crash consistency based on back pointers. Read it for the exciting details!*

[C+13] "Optimistic Crash Consistency"

Vijay Chidambaram, Thanu S. Pillai, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau  
SOSP '13, Nemacolin Woodlands Resort, PA, November 2013

*Our work on a more optimistic and higher performance journaling protocol. For workloads that call `fsync()` a lot, performance can be greatly improved.*

[GP94] "Metadata Update Performance in File Systems"

Gregory R. Ganger and Yale N. Patt

OSDI '94

*A clever paper about using careful ordering of writes as the main way to achieve consistency. Implemented later in BSD-based systems.*

[G+08] "SQCK: A Declarative File System Checker"

Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau  
OSDI '08, San Diego, California

*Our own paper on a new and better way to build a file system checker using SQL queries. We also show some problems with the existing checker, finding numerous bugs and odd behaviors, a direct result of the complexity of `fsck`.*

[H87] "Reimplementing the Cedar File System Using Logging and Group Commit"

Robert Hagmann

SOSP '87, Austin, Texas, November 1987

*The first work (that we know of) that applied write-ahead logging (a.k.a. journaling) to a file system.*

[M+13] "ffsck: The Fast File System Checker"

Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

FAST '13, San Jose, California, February 2013

*A recent paper of ours detailing how to make `fsck` an order of magnitude faster. Some of the ideas have already been incorporated into the BSD file system checker [MK96] and are deployed today.*

[MK96] "Fsck - The UNIX File System Check Program"

Marshall Kirk McKusick and T. J. Kowalski

Revised in 1996

*Describes the first comprehensive file-system checking tool, the eponymous `fsck`. Written by some of the same people who brought you FFS.*

[MJLF84] "A Fast File System for UNIX"

Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry

ACM Transactions on Computing Systems.

August 1984, Volume 2:3

*You already know enough about FFS, right? But yeah, it is OK to reference papers like this more than once in a book.*

[P+05] "IRON File Systems"

Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau  
SOSP '05, Brighton, England, October 2005

*A paper mostly focused on studying how file systems react to disk failures. Towards the end, we introduce a transaction checksum to speed up logging, which was eventually adopted into Linux ext4.*

[PAA05] "Analysis and Evolution of Journaling File Systems"

Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau  
USENIX '05, Anaheim, California, April 2005

*An early paper we wrote analyzing how journaling file systems work.*

[R+11] "Coerced Cache Eviction and Discreet-Mode Journaling"

Abhishek Rajimwale, Vijay Chidambaram, Deepak Ramamurthy,  
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau  
DSN '11, Hong Kong, China, June 2011

*Our own paper on the problem of disks that buffer writes in a memory cache instead of forcing them to disk, even when explicitly told not to do that! Our solution to overcome this problem: if you want A to be written to disk before B, first write A, then send a lot of "dummy" writes to disk, hopefully causing A to be forced to disk to make room for them in the cache. A neat if impractical solution.*

[T98] "Journaling the Linux ext2fs File System"

Stephen C. Tweedie

The Fourth Annual Linux Expo, May 1998

*Tweedie did much of the heavy lifting in adding journaling to the Linux ext2 file system; the result, not surprisingly, is called ext3. Some nice design decisions include the strong focus on backwards compatibility, e.g., you can just add a journaling file to an existing ext2 file system and then mount it as an ext3 file system.*

[T00] "EXT3, Journaling Filesystem"

Stephen Tweedie

Talk at the Ottawa Linux Symposium, July 2000

[olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html](http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html)

*A transcript of a talk given by Tweedie on ext3.*

[T01] "The Linux ext2 File System"

Theodore Ts'o, June, 2001.

Available: <http://e2fsprogs.sourceforge.net/ext2.html>

*A simple Linux file system based on the ideas found in FFS. For a while it was quite heavily used; now it is really just in the kernel as an example of a simple file system.*

## Log-structured File Systems

In the early 90's, a group at Berkeley led by Professor John Ousterhout and graduate student Mendel Rosenblum developed a new file system known as the log-structured file system [RO91]. Their motivation to do so was based on the following observations:

- **System memories are growing:** As memory gets bigger, more data can be cached in memory. As more data is cached, disk traffic increasingly consists of writes, as reads are serviced by the cache. Thus, file system performance is largely determined by its write performance.
- **There is a large gap between random I/O performance and sequential I/O performance:** Hard-drive transfer bandwidth has increased a great deal over the years [P98]; as more bits are packed into the surface of a drive, the bandwidth when accessing said bits increases. Seek and rotational delay costs, however, have decreased slowly; it is challenging to make cheap and small motors spin the platters faster or move the disk arm more quickly. Thus, if you are able to use disks in a sequential manner, you gain a sizeable performance advantage over approaches that cause seeks and rotations.
- **Existing file systems perform poorly on many common workloads:** For example, FFS [MJLF84] would perform a large number of writes to create a new file of size one block: one for a new inode, one to update the inode bitmap, one to the directory data block that the file is in, one to the directory inode to update it, one to the new data block that is a part of the new file, and one to the data bitmap to mark the data block as allocated. Thus, although FFS places all of these blocks within the same block group, FFS incurs many short seeks and subsequent rotational delays and thus performance falls far short of peak sequential bandwidth.
- **File systems are not RAID-aware:** For example, both RAID-4 and RAID-5 have the **small-write problem** where a logical write to a single block causes 4 physical I/Os to take place. Existing file systems do not try to avoid this worst-case RAID writing behavior.

**TIP: DETAILS MATTER**

All interesting systems are comprised of a few general ideas and a number of details. Sometimes, when you are learning about these systems, you think to yourself “Oh, I get the general idea; the rest is just details,” and you use this to only half-learn how things really work. Don’t do this! Many times, the details are critical. As we’ll see with LFS, the general idea is easy to understand, but to really build a working system, you have to think through *all* of the tricky cases.

An ideal file system would thus focus on write performance, and try to make use of the sequential bandwidth of the disk. Further, it would perform well on common workloads that not only write out data but also update on-disk metadata structures frequently. Finally, it would work well on RAIDs as well as single disks.

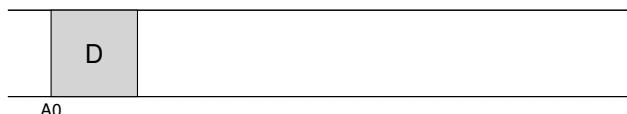
The new type of file system Rosenblum and Ousterhout introduced was called **LFS**, short for the **Log-structured File System**. When writing to disk, LFS first buffers all updates (including metadata!) in an in-memory **segment**; when the segment is full, it is written to disk in one long, sequential transfer to an unused part of the disk. LFS never overwrites existing data, but rather *always* writes segments to free locations. Because segments are large, the disk is used efficiently, and performance of the file system approaches its zenith.

**THE CRUX:****HOW TO MAKE ALL WRITES SEQUENTIAL WRITES?**

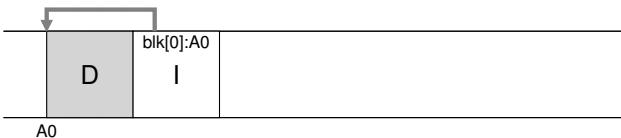
How can a file system transform all writes into sequential writes? For reads, this task is impossible, as the desired block to be read may be anywhere on disk. For writes, however, the file system always has a choice, and it is exactly this choice we hope to exploit.

### 43.1 Writing To Disk Sequentially

We thus have our first challenge: how do we transform all updates to file-system state into a series of sequential writes to disk? To understand this better, let’s use a simple example. Imagine we are writing a data block *D* to a file. Writing the data block to disk might result in the following on-disk layout, with *D* written at disk address *A*<sub>0</sub>:



However, when a user writes a data block, it is not only data that gets written to disk; there is also other **metadata** that needs to be updated. In this case, let's also write the **inode** (*I*) of the file to disk, and have it point to the data block *D*. When written to disk, the data block and inode would look something like this (note that the inode looks as big as the data block, which generally isn't the case; in most systems, data blocks are 4 KB in size, whereas an inode is much smaller, around 128 bytes):



This basic idea, of simply writing all updates (such as data blocks, inodes, etc.) to the disk sequentially, sits at the heart of LFS. If you understand this, you get the basic idea. But as with all complicated systems, the devil is in the details.

## 43.2 Writing Sequentially And Effectively

Unfortunately, writing to disk sequentially is not (alone) enough to guarantee efficient writes. For example, imagine if we wrote a single block to address *A*, at time *T*. We then wait a little while, and write to the disk at address *A* + 1 (the next block address in sequential order), but at time *T* +  $\delta$ . In-between the first and second writes, unfortunately, the disk has rotated; when you issue the second write, it will thus wait for most of a rotation before being committed (specifically, if the rotation takes time  $T_{rotation}$ , the disk will wait  $T_{rotation} - \delta$  before it can commit the second write to the disk surface). And thus you can hopefully see that simply writing to disk in sequential order is not enough to achieve peak performance; rather, you must issue a large number of *contiguous* writes (or one large write) to the drive in order to achieve good write performance.

To achieve this end, LFS uses an ancient technique known as **write buffering**<sup>1</sup>. Before writing to the disk, LFS keeps track of updates in memory; when it has received a sufficient number of updates, it writes them to disk all at once, thus ensuring efficient use of the disk.

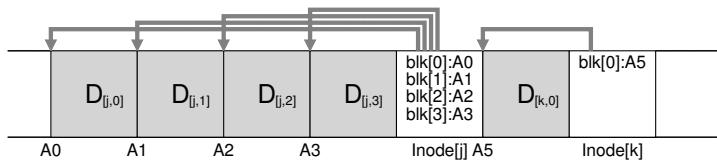
The large chunk of updates LFS writes at one time is referred to by the name of a **segment**. Although this term is over-used in computer systems, here it just means a large-ish chunk which LFS uses to group writes. Thus, when writing to disk, LFS buffers updates in an in-memory

---

<sup>1</sup>Indeed, it is hard to find a good citation for this idea, since it was likely invented by many and very early on in the history of computing. For a study of the benefits of write buffering, see Solworth and Orji [SO90]; to learn about its potential harms, see Mogul [M94].

segment, and then writes the segment all at once to the disk. As long as the segment is large enough, these writes will be efficient.

Here is an example, in which LFS buffers two sets of updates into a small segment; actual segments are larger (a few MB). The first update is of four block writes to file  $j$ ; the second is one block being added to file  $k$ . LFS then commits the entire segment of seven blocks to disk at once. The resulting on-disk layout of these blocks is as follows:



### 43.3 How Much To Buffer?

This raises the following question: how many updates should LFS buffer before writing to disk? The answer, of course, depends on the disk itself, specifically how high the positioning overhead is in comparison to the transfer rate; see the FFS chapter for a similar analysis.

For example, assume that positioning (i.e., rotation and seek overheads) before each write takes roughly  $T_{position}$  seconds. Assume further that the disk transfer rate is  $R_{peak}$  MB/s. How much should LFS buffer before writing when running on such a disk?

The way to think about this is that every time you write, you pay a fixed overhead of the positioning cost. Thus, how much do you have to write in order to **amortize** that cost? The more you write, the better (obviously), and the closer you get to achieving peak bandwidth.

To obtain a concrete answer, let's assume we are writing out  $D$  MB. The time to write out this chunk of data ( $T_{write}$ ) is the positioning time  $T_{position}$  plus the time to transfer  $D$  ( $\frac{D}{R_{peak}}$ ), or:

$$T_{write} = T_{position} + \frac{D}{R_{peak}} \quad (43.1)$$

And thus the effective *rate* of writing ( $R_{effective}$ ), which is just the amount of data written divided by the total time to write it, is:

$$R_{effective} = \frac{D}{T_{write}} = \frac{D}{T_{position} + \frac{D}{R_{peak}}} \quad (43.2)$$

What we're interested in is getting the effective rate ( $R_{effective}$ ) close to the peak rate. Specifically, we want the effective rate to be some fraction  $F$  of the peak rate, where  $0 < F < 1$  (a typical  $F$  might be 0.9, or 90% of the peak rate). In mathematical form, this means we want  $R_{effective} = F \times R_{peak}$ .

At this point, we can solve for  $D$ :

$$R_{\text{effective}} = \frac{D}{T_{\text{position}} + \frac{D}{R_{\text{peak}}}} = F \times R_{\text{peak}} \quad (43.3)$$

$$D = F \times R_{\text{peak}} \times \left( T_{\text{position}} + \frac{D}{R_{\text{peak}}} \right) \quad (43.4)$$

$$D = (F \times R_{\text{peak}} \times T_{\text{position}}) + (F \times R_{\text{peak}} \times \frac{D}{R_{\text{peak}}}) \quad (43.5)$$

$$D = \frac{F}{1 - F} \times R_{\text{peak}} \times T_{\text{position}} \quad (43.6)$$

Let's do an example, with a disk with a positioning time of 10 milliseconds and peak transfer rate of 100 MB/s; assume we want an effective bandwidth of 90% of peak ( $F = 0.9$ ). In this case,  $D = \frac{0.9}{0.1} \times 100 \text{ MB/s} \times 0.01 \text{ seconds} = 9 \text{ MB}$ . Try some different values to see how much we need to buffer in order to approach peak bandwidth. How much is needed to reach 95% of peak? 99%?

#### 43.4 Problem: Finding Inodes

To understand how we find an inode in LFS, let us briefly review how to find an inode in a typical UNIX file system. In a typical file system such as FFS, or even the old UNIX file system, finding inodes is easy, because they are organized in an array and placed on disk at fixed locations.

For example, the old UNIX file system keeps all inodes at a fixed portion of the disk. Thus, given an inode number and the start address, to find a particular inode, you can calculate its exact disk address simply by multiplying the inode number by the size of an inode, and adding that to the start address of the on-disk array; array-based indexing, given an inode number, is fast and straightforward.

Finding an inode given an inode number in FFS is only slightly more complicated, because FFS splits up the inode table into chunks and places a group of inodes within each cylinder group. Thus, one must know how big each chunk of inodes is and the start addresses of each. After that, the calculations are similar and also easy.

In LFS, life is more difficult. Why? Well, we've managed to scatter the inodes all throughout the disk! Worse, we never overwrite in place, and thus the latest version of an inode (i.e., the one we want) keeps moving.

#### 43.5 Solution Through Indirection: The Inode Map

To remedy this, the designers of LFS introduced a **level of indirection** between inode numbers and the inodes through a data structure called the **inode map (imap)**. The imap is a structure that takes an inode number as input and produces the disk address of the most recent version of the

**TIP: USE A LEVEL OF INDIRECTION**

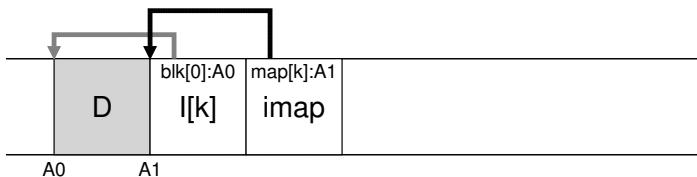
People often say that the solution to all problems in Computer Science is simply a **level of indirection**. This is clearly not true; it is just the solution to *most* problems (yes, this is still too strong of a comment, but you get the point). You certainly can think of every virtualization we have studied, e.g., virtual memory, or the notion of a file, as simply a level of indirection. And certainly the inode map in LFS is a virtualization of inode numbers. Hopefully you can see the great power of indirection in these examples, allowing us to freely move structures around (such as pages in the VM example, or inodes in LFS) without having to change every reference to them. Of course, indirection can have a downside too: **extra overhead**. So next time you have a problem, try solving it with indirection, but make sure to think about the overheads of doing so first. As Wheeler famously said, “All problems in computer science can be solved by another level of indirection, except of course for the problem of too many indirections.”

inode. Thus, you can imagine it would often be implemented as a simple *array*, with 4 bytes (a disk pointer) per entry. Any time an inode is written to disk, the imap is updated with its new location.

The imap, unfortunately, needs to be kept persistent (i.e., written to disk); doing so allows LFS to keep track of the locations of inodes across crashes, and thus operate as desired. Thus, a question: where should the imap reside on disk?

It could live on a fixed part of the disk, of course. Unfortunately, as it gets updated frequently, this would then require updates to file structures to be followed by writes to the imap, and hence performance would suffer (i.e., there would be more disk seeks, between each update and the fixed location of the imap).

Instead, LFS places chunks of the inode map right next to where it is writing all of the other new information. Thus, when appending a data block to a file  $k$ , LFS actually writes the new data block, its inode, and a piece of the inode map all together onto the disk, as follows:



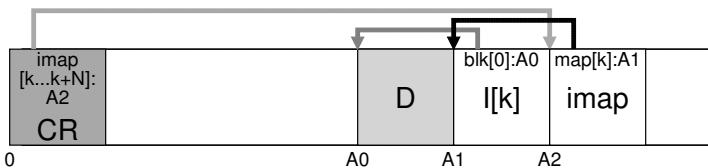
In this picture, the piece of the imap array stored in the block marked *imap* tells LFS that the inode  $k$  is at disk address  $A1$ ; this inode, in turn, tells LFS that its data block  $D$  is at address  $A0$ .

## 43.6 Completing The Solution: The Checkpoint Region

The clever reader (that's you, right?) might have noticed a problem here. How do we find the inode map, now that pieces of it are also now spread across the disk? In the end, there is no magic: the file system must have *some* fixed and known location on disk to begin a file lookup.

LFS has just such a fixed place on disk for this, known as the **checkpoint region (CR)**. The checkpoint region contains pointers to (i.e., addresses of) the latest pieces of the inode map, and thus the inode map pieces can be found by reading the CR first. Note the checkpoint region is only updated periodically (say every 30 seconds or so), and thus performance is not ill-affected. Thus, the overall structure of the on-disk layout contains a checkpoint region (which points to the latest pieces of the inode map); the inode map pieces each contain addresses of the inodes; the inodes point to files (and directories) just like typical UNIX file systems.

Here is an example of the checkpoint region (note it is all the way at the beginning of the disk, at address 0), and a single imap chunk, inode, and data block. A real file system would of course have a much bigger CR (indeed, it would have two, as we'll come to understand later), many imap chunks, and of course many more inodes, data blocks, etc.



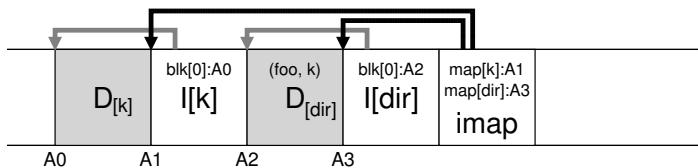
## 43.7 Reading A File From Disk: A Recap

To make sure you understand how LFS works, let us now walk through what must happen to read a file from disk. Assume we have nothing in memory to begin. The first on-disk data structure we must read is the checkpoint region. The checkpoint region contains pointers (i.e., disk addresses) to the entire inode map, and thus LFS then reads in the entire inode map and caches it in memory. After this point, when given an inode number of a file, LFS simply looks up the inode-number to inode-disk-address mapping in the imap, and reads in the most recent version of the inode. To read a block from the file, at this point, LFS proceeds exactly as a typical UNIX file system, by using direct pointers or indirect pointers or doubly-indirect pointers as need be. In the common case, LFS should perform the same number of I/Os as a typical file system when reading a file from disk; the entire imap is cached and thus the extra work LFS does during a read is to look up the inode's address in the imap.

### 43.8 What About Directories?

Thus far, we've simplified our discussion a bit by only considering inodes and data blocks. However, to access a file in a file system (such as `/home/remzi/foo`, one of our favorite fake file names), some directories must be accessed too. So how does LFS store directory data?

Fortunately, directory structure is basically identical to classic UNIX file systems, in that a directory is just a collection of (name, inode number) mappings. For example, when creating a file on disk, LFS must both write a new inode, some data, as well as the directory data and its inode that refer to this file. Remember that LFS will do so sequentially on the disk (after buffering the updates for some time). Thus, creating a file `foo` in a directory would lead to the following new structures on disk:



The piece of the inode map contains the information for the location of both the directory file `dir` as well as the newly-created file `f`. Thus, when accessing file `foo` (with inode number  $k$ ), you would first look in the inode map (usually cached in memory) to find the location of the inode of directory `dir` ( $A_3$ ); you then read the directory inode, which gives you the location of the directory data ( $A_2$ ); reading this data block gives you the name-to-inode-number mapping of  $(\text{foo}, k)$ . You then consult the inode map again to find the location of inode number  $k$  ( $A_1$ ), and finally read the desired data block at address  $A_0$ .

There is one other serious problem in LFS that the inode map solves, known as the **recursive update problem** [Z+12]. The problem arises in any file system that never updates in place (such as LFS), but rather moves updates to new locations on the disk.

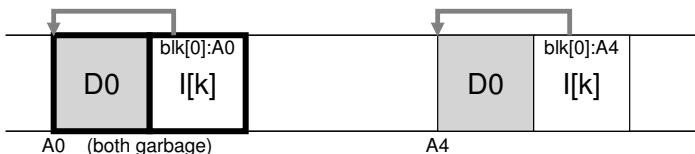
Specifically, whenever an inode is updated, its location on disk changes. If we hadn't been careful, this would have also entailed an update to the directory that points to this file, which then would have mandated a change to the parent of that directory, and so on, all the way up the file system tree.

LFS cleverly avoids this problem with the inode map. Even though the location of an inode may change, the change is never reflected in the directory itself; rather, the imap structure is updated while the directory holds the same name-to-inumber mapping. Thus, through indirection, LFS avoids the recursive update problem.

### 43.9 A New Problem: Garbage Collection

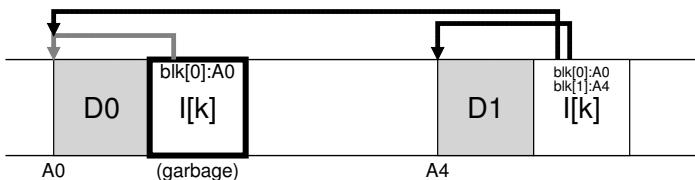
You may have noticed another problem with LFS; it repeatedly writes the latest version of a file (including its inode and data) to new locations on disk. This process, while keeping writes efficient, implies that LFS leaves old versions of file structures scattered throughout the disk. We (rather unceremoniously) call these old versions **garbage**.

For example, let's imagine the case where we have an existing file referred to by inode number  $k$ , which points to a single data block  $D_0$ . We now update that block, generating both a new inode and a new data block. The resulting on-disk layout of LFS would look something like this (note we omit the imap and other structures for simplicity; a new chunk of imap would also have to be written to disk to point to the new inode):



In the diagram, you can see that both the inode and data block have two versions on disk, one old (the one on the left) and one current and thus live (the one on the right). By the simple act of (logically) updating a data block, a number of new structures must be persisted by LFS, thus leaving old versions of said blocks on the disk.

As another example, imagine we instead append a block to that original file  $k$ . In this case, a new version of the inode is generated, but the old data block is still pointed to by the inode. Thus, it is still live and very much part of the current file system:



So what should we do with these older versions of inodes, data blocks, and so forth? One could keep those older versions around and allow users to restore old file versions (for example, when they accidentally overwrite or delete a file, it could be quite handy to do so); such a file system is known as a **versioning file system** because it keeps track of the different versions of a file.

However, LFS instead keeps only the latest live version of a file; thus (in the background), LFS must periodically find these old dead versions of file data, inodes, and other structures, and **clean** them; cleaning should

thus make blocks on disk free again for use in a subsequent writes. Note that the process of cleaning is a form of **garbage collection**, a technique that arises in programming languages that automatically free unused memory for programs.

Earlier we discussed segments as important as they are the mechanism that enables large writes to disk in LFS. As it turns out, they are also quite integral to effective cleaning. Imagine what would happen if the LFS cleaner simply went through and freed single data blocks, inodes, etc., during cleaning. The result: a file system with some number of free **holes** mixed between allocated space on disk. Write performance would drop considerably, as LFS would not be able to find a large contiguous region to write to disk sequentially and with high performance.

Instead, the LFS cleaner works on a segment-by-segment basis, thus clearing up large chunks of space for subsequent writing. The basic cleaning process works as follows. Periodically, the LFS cleaner reads in a number of old (partially-used) segments, determines which blocks are live within these segments, and then write out a new set of segments with just the live blocks within them, freeing up the old ones for writing. Specifically, we expect the cleaner to read in  $M$  existing segments, **compact** their contents into  $N$  new segments (where  $N < M$ ), and then write the  $N$  segments to disk in new locations. The old  $M$  segments are then freed and can be used by the file system for subsequent writes.

We are now left with two problems, however. The first is mechanism: how can LFS tell which blocks within a segment are live, and which are dead? The second is policy: how often should the cleaner run, and which segments should it pick to clean?

### 43.10 Determining Block Liveness

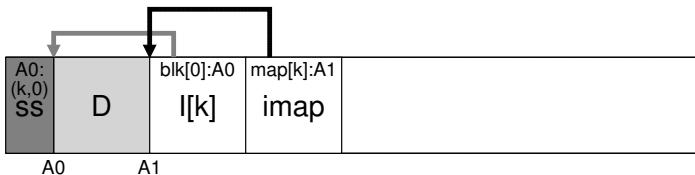
We address the mechanism first. Given a data block  $D$  within an on-disk segment  $S$ , LFS must be able to determine whether  $D$  is live. To do so, LFS adds a little extra information to each segment that describes each block. Specifically, LFS includes, for each data block  $D$ , its inode number (which file it belongs to) and its offset (which block of the file this is). This information is recorded in a structure at the head of the segment known as the **segment summary block**.

Given this information, it is straightforward to determine whether a block is live or dead. For a block  $D$  located on disk at address  $A$ , look in the segment summary block and find its inode number  $N$  and offset  $T$ . Next, look in the imap to find where  $N$  lives and read  $N$  from disk (perhaps it is already in memory, which is even better). Finally, using the offset  $T$ , look in the inode (or some indirect block) to see where the inode thinks the  $T$ th block of this file is on disk. If it points exactly to disk address  $A$ , LFS can conclude that the block  $D$  is live. If it points anywhere else, LFS can conclude that  $D$  is not in use (i.e., it is dead) and thus know that this version is no longer needed. A pseudocode summary of this

process is shown here:

```
(N, T) = SegmentSummary[A];
inode = Read(imap[N]);
if (inode[T] == A)
    // block D is alive
else
    // block D is garbage
```

Here is a diagram depicting the mechanism, in which the segment summary block (marked *SS*) records that the data block at address *A*<sub>0</sub> is actually a part of file *k* at offset 0. By checking the imap for *k*, you can find the inode, and see that it does indeed point to that location.



There are some shortcuts LFS takes to make the process of determining liveness more efficient. For example, when a file is truncated or deleted, LFS increases its **version number** and records the new version number in the imap. By also recording the version number in the on-disk segment, LFS can short circuit the longer check described above simply by comparing the on-disk version number with a version number in the imap, thus avoiding extra reads.

### 43.11 A Policy Question: Which Blocks To Clean, And When?

On top of the mechanism described above, LFS must include a set of policies to determine both when to clean and which blocks are worth cleaning. Determining when to clean is easier; either periodically, during idle time, or when you have to because the disk is full.

Determining which blocks to clean is more challenging, and has been the subject of many research papers. In the original LFS paper [RO91], the authors describe an approach which tries to segregate *hot* and *cold* segments. A hot segment is one in which the contents are being frequently over-written; thus, for such a segment, the best policy is to wait a long time before cleaning it, as more and more blocks are getting over-written (in new segments) and thus being freed for use. A cold segment, in contrast, may have a few dead blocks but the rest of its contents are relatively stable. Thus, the authors conclude that one should clean cold segments sooner and hot segments later, and develop a heuristic that does exactly that. However, as with most policies, this policy isn't perfect; later approaches show how to do better [MR+97].

### 43.12 Crash Recovery And The Log

One final problem: what happens if the system crashes while LFS is writing to disk? As you may recall in the previous chapter about journaling, crashes during updates are tricky for file systems, and thus something LFS must consider as well.

During normal operation, LFS buffers writes in a segment, and then (when the segment is full, or when some amount of time has elapsed), writes the segment to disk. LFS organizes these writes in a **log**, i.e., the checkpoint region points to a head and tail segment, and each segment points to the next segment to be written. LFS also periodically updates the checkpoint region. Crashes could clearly happen during either of these operations (write to a segment, write to the CR). So how does LFS handle crashes during writes to these structures?

Let's cover the second case first. To ensure that the CR update happens atomically, LFS actually keeps two CRs, one at either end of the disk, and writes to them alternately. LFS also implements a careful protocol when updating the CR with the latest pointers to the inode map and other information; specifically, it first writes out a header (with timestamp), then the body of the CR, and then finally one last block (also with a timestamp). If the system crashes during a CR update, LFS can detect this by seeing an inconsistent pair of timestamps. LFS will always choose to use the most recent CR that has consistent timestamps, and thus consistent update of the CR is achieved.

Let's now address the first case. Because LFS writes the CR every 30 seconds or so, the last consistent snapshot of the file system may be quite old. Thus, upon reboot, LFS can easily recover by simply reading in the checkpoint region, the imap pieces it points to, and subsequent files and directories; however, the last many seconds of updates would be lost.

To improve upon this, LFS tries to rebuild many of those segments through a technique known as **roll forward** in the database community. The basic idea is to start with the last checkpoint region, find the end of the log (which is included in the CR), and then use that to read through the next segments and see if there are any valid updates within it. If there are, LFS updates the file system accordingly and thus recovers much of the data and metadata written since the last checkpoint. See Rosenblum's award-winning dissertation for details [R92].

### 43.13 Summary

LFS introduces a new approach to updating the disk. Instead of overwriting files in places, LFS always writes to an unused portion of the disk, and then later reclaims that old space through cleaning. This approach, which in database systems is called **shadow paging** [L77] and in file-system-speak is sometimes called **copy-on-write**, enables highly efficient writing, as LFS can gather all updates into an in-memory segment and then write them out together sequentially.

**TIP: TURN FLAWS INTO VIRTUES**

Whenever your system has a fundamental flaw, see if you can turn it around into a feature or something useful. NetApp's WAFL does this with old file contents; by making old versions available, WAFL no longer has to worry about cleaning quite so often (though it does delete old versions, eventually, in the background), and thus provides a cool feature and removes much of the LFS cleaning problem all in one wonderful twist. Are there other examples of this in systems? Undoubtedly, but you'll have to think of them yourself, because this chapter is over with a capital "O". Over. Done. Kaput. We're out. Peace!

The downside to this approach is that it generates garbage; old copies of the data are scattered throughout the disk, and if one wants to reclaim such space for subsequent usage, one must clean old segments periodically. Cleaning became the focus of much controversy in LFS, and concerns over cleaning costs [SS+95] perhaps limited LFS's initial impact on the field. However, some modern commercial file systems, including NetApp's **WAFL** [HLM94], Sun's **ZFS** [B07], and Linux **btrfs** [M07], and even modern **flash-based SSDs** [AD14], adopt a similar copy-on-write approach to writing to disk, and thus the intellectual legacy of LFS lives on in these modern file systems. In particular, WAFL got around cleaning problems by turning them into a feature; by providing old versions of the file system via **snapshots**, users could access old files whenever they deleted current ones accidentally.

## References

- [AD14] "Operating Systems: Three Easy Pieces"  
*Chapter: Flash-based Solid State Drives*  
 Remzi Arpacı-Dusseau and Andrea Arpacı-Dusseau  
*A bit gauche to refer you to another chapter in this very book, but who are we to judge?*
- [B07] "ZFS: The Last Word in File Systems"  
 Jeff Bonwick and Bill Moore  
 Copy Available: <http://www.ostep.org/Citations/zfs.last.pdf>  
*Slides on ZFS; unfortunately, there is no great ZFS paper (yet). Maybe you will write one, so we can cite it here?*
- [HLM94] "File System Design for an NFS File Server Appliance"  
 Dave Hitz, James Lau, Michael Malcolm  
 USENIX Spring '94  
*WAFL takes many ideas from LFS and RAID and puts it into a high-speed NFS appliance for the multi-billion dollar storage company NetApp.*
- [L77] "Physical Integrity in a Large Segmented Database"  
 R. Lorie  
 ACM Transactions on Databases, 1977, Volume 2:1, pages 91-104  
*The original idea of shadow paging is presented here.*
- [M07] "The Btrfs Filesystem"  
 Chris Mason  
 September 2007  
 Available: [oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf](http://oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf)  
*A recent copy-on-write Linux file system, slowly gaining in importance and usage.*
- [MJLF84] "A Fast File System for UNIX"  
 Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry  
 ACM TOCS, August, 1984, Volume 2, Number 3  
*The original FFS paper; see the chapter on FFS for more details.*
- [MR+97] "Improving the Performance of Log-structured File Systems with Adaptive Methods"  
 Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello,  
 Randolph Y. Wang, Thomas E. Anderson  
 SOSP 1997, pages 238-251, October, Saint Malo, France  
*A more recent paper detailing better policies for cleaning in LFS.*
- [M94] "A Better Update Policy"  
 Jeffrey C. Mogul  
 USENIX ATC '94, June 1994  
*In this paper, Mogul finds that read workloads can be harmed by buffering writes for too long and then sending them to the disk in a big burst. Thus, he recommends sending writes more frequently and in smaller batches.*
- [P98] "Hardware Technology Trends and Database Opportunities"  
 David A. Patterson  
 ACM SIGMOD '98 Keynote Address, Presented June 3, 1998, Seattle, Washington  
 Available: <http://www.cs.berkeley.edu/~pattrsn/talks/keynote.html>  
*A great set of slides on technology trends in computer systems. Hopefully, Patterson will create another of these sometime soon.*

[RO91] "Design and Implementation of the Log-structured File System"

Mendel Rosenblum and John Ousterhout

SOSP '91, Pacific Grove, CA, October 1991

*The original SOSP paper about LFS, which has been cited by hundreds of other papers and inspired many real systems.*

[R92] "Design and Implementation of the Log-structured File System"

Mendel Rosenblum

<http://www.eecs.berkeley.edu/Pubs/TechRpts/1992/CSD-92-696.pdf>

*The award-winning dissertation about LFS, with many of the details missing from the paper.*

[SS+95] "File system logging versus clustering: a performance comparison"

Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, Venkata Padmanabhan

USENIX 1995 Technical Conference, New Orleans, Louisiana, 1995

*A paper that showed the LFS performance sometimes has problems, particularly for workloads with many calls to `fsync()` (such as database workloads). The paper was controversial at the time.*

[SO90] "Write-Only Disk Caches"

Jon A. Solworth, Cyril U. Orji

SIGMOD '90, Atlantic City, New Jersey, May 1990

*An early study of write buffering and its benefits. However, buffering for too long can be harmful: see Mogul [M94] for details.*

[Z+12] "De-indirection for Flash-based SSDs with Nameless Writes"

Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

FAST '13, San Jose, California, February 2013

*Our paper on a new way to build flash-based storage devices. Because FTLs (flash-translation layers) are usually built in a log-structured style, some of the same issues arise in flash-based devices that do in LFS. In this case, it is the recursive update problem, which LFS solves neatly with an imap. A similar structure exists in most SSDs.*

## Data Integrity and Protection

Beyond the basic advances found in the file systems we have studied thus far, a number of features are worth studying. In this chapter, we focus on reliability once again (having previously studied storage system reliability in the RAID chapter). Specifically, how should a file system or storage system ensure that data is safe, given the unreliable nature of modern storage devices?

This general area is referred to as **data integrity** or **data protection**. Thus, we will now investigate techniques used to ensure that the data you put into your storage system is the same when the storage system returns it to you.

### CRUX: HOW TO ENSURE DATA INTEGRITY

How should systems ensure that the data written to storage is protected? What techniques are required? How can such techniques be made efficient, with both low space and time overheads?

### 44.1 Disk Failure Modes

As you learned in the chapter about RAID, disks are not perfect, and can fail (on occasion). In early RAID systems, the model of failure was quite simple: either the entire disk is working, or it fails completely, and the detection of such a failure is straightforward. This **fail-stop** model of disk failure makes building RAID relatively simple [S90].

What you didn't learn is about all of the other types of failure modes modern disks exhibit. Specifically, as Bairavasundaram et al. studied in great detail [B+07, B+08], modern disks will occasionally seem to be mostly working but have trouble successfully accessing one or more blocks. Specifically, two types of single-block failures are common and worthy of consideration: **latent-sector errors (LSEs)** and **block corruption**. We'll now discuss each in more detail.

	Cheap	Costly
LSEs	9.40%	1.40%
Corruption	0.50%	0.05%

Figure 44.1: Frequency Of LSEs And Block Corruption

LSEs arise when a disk sector (or group of sectors) has been damaged in some way. For example, if the disk head touches the surface for some reason (a **head crash**, something which shouldn't happen during normal operation), it may damage the surface, making the bits unreadable. Cosmic rays can also flip bits, leading to incorrect contents. Fortunately, in-disk **error correcting codes** (ECC) are used by the drive to determine whether the on-disk bits in a block are good, and in some cases, to fix them; if they are not good, and the drive does not have enough information to fix the error, the disk will return an error when a request is issued to read them.

There are also cases where a disk block becomes **corrupt** in a way not detectable by the disk itself. For example, buggy disk firmware may write a block to the wrong location; in such a case, the disk ECC indicates the block contents are fine, but from the client's perspective the wrong block is returned when subsequently accessed. Similarly, a block may get corrupted when it is transferred from the host to the disk across a faulty bus; the resulting corrupt data is stored by the disk, but it is not what the client desires. These types of faults are particularly insidious because they are **silent faults**; the disk gives no indication of the problem when returning the faulty data.

Prabhakaran et al. describes this more modern view of disk failure as the **fail-partial** disk failure model [P+05]. In this view, disks can still fail in their entirety (as was the case in the traditional fail-stop model); however, disks can also seemingly be working and have one or more blocks become inaccessible (i.e., LSEs) or hold the wrong contents (i.e., corruption). Thus, when accessing a seemingly-working disk, once in a while it may either return an error when trying to read or write a given block (a non-silent partial fault), and once in a while it may simply return the wrong data (a silent partial fault).

Both of these types of faults are somewhat rare, but just how rare? Figure 44.1 summarizes some of the findings from the two Bairavasundaram studies [B+07,B+08].

The figure shows the percent of drives that exhibited at least one LSE or block corruption over the course of the study (about 3 years, over 1.5 million disk drives). The figure further sub-divides the results into “cheap” drives (usually SATA drives) and “costly” drives (usually SCSI or FibreChannel). As you can see, while buying better drives reduces the frequency of both types of problem (by about an order of magnitude), they still happen often enough that you need to think carefully about how to handle them in your storage system.

Some additional findings about LSEs are:

- Costly drives with more than one LSE are as likely to develop additional errors as cheaper drives
- For most drives, annual error rate increases in year two
- The number of LSEs increase with disk size
- Most disks with LSEs have less than 50
- Disks with LSEs are more likely to develop additional LSEs
- There exists a significant amount of spatial and temporal locality
- Disk scrubbing is useful (most LSEs were found this way)

Some findings about corruption:

- Chance of corruption varies greatly across different drive models within the same drive class
- Age effects are different across models
- Workload and disk size have little impact on corruption
- Most disks with corruption only have a few corruptions
- Corruption is not independent within a disk or across disks in RAID
- There exists spatial locality, and some temporal locality
- There is a weak correlation with LSEs

To learn more about these failures, you should likely read the original papers [B+07,B+08]. But hopefully the main point should be clear: if you really wish to build a reliable storage system, you must include machinery to detect and recover from both LSEs and block corruption.

## 44.2 Handling Latent Sector Errors

Given these two new modes of partial disk failure, we should now try to see what we can do about them. Let's first tackle the easier of the two, namely latent sector errors.

### CRUX: HOW TO HANDLE LATENT SECTOR ERRORS

How should a storage system handle latent sector errors? How much extra machinery is needed to handle this form of partial failure?

As it turns out, latent sector errors are rather straightforward to handle, as they are (by definition) easily detected. When a storage system tries to access a block, and the disk returns an error, the storage system should simply use whatever redundancy mechanism it has to return the correct data. In a mirrored RAID, for example, the system should access the alternate copy; in a RAID-4 or RAID-5 system based on parity, the system should reconstruct the block from the other blocks in the parity group. Thus, easily detected problems such as LSEs are readily recovered through standard redundancy mechanisms.

The growing prevalence of LSEs has influenced RAID designs over the years. One particularly interesting problem arises in RAID-4/5 systems when both full-disk faults and LSEs occur in tandem. Specifically, when an entire disk fails, the RAID tries to **reconstruct** the disk (say, onto a hot spare) by reading through all of the other disks in the parity group and recomputing the missing values. If, during reconstruction, an LSE is encountered on any one of the other disks, we have a problem: the reconstruction cannot successfully complete.

To combat this issue, some systems add an extra degree of redundancy. For example, NetApp's **RAID-DP** has the equivalent of two parity disks instead of one [C+04]. When an LSE is discovered during reconstruction, the extra parity helps to reconstruct the missing block. As always, there is a cost, in that maintaining two parity blocks for each stripe is more costly; however, the log-structured nature of the NetApp **WAFL** file system mitigates that cost in many cases [HLM94]. The remaining cost is space, in the form of an extra disk for the second parity block.

### 44.3 Detecting Corruption: The Checksum

Let's now tackle the more challenging problem, that of silent failures via data corruption. How can we prevent users from getting bad data when corruption arises, and thus leads to disks returning bad data?

#### CRUX: HOW TO PRESERVE DATA INTEGRITY DESPITE CORRUPTION

Given the silent nature of such failures, what can a storage system do to detect when corruption arises? What techniques are needed? How can one implement them efficiently?

Unlike latent sector errors, *detection* of corruption is a key problem. How can a client tell that a block has gone bad? Once it is known that a particular block is bad, *recovery* is the same as before: you need to have some other copy of the block around (and hopefully, one that is not corrupt!). Thus, we focus here on detection techniques.

The primary mechanism used by modern storage systems to preserve data integrity is called the **checksum**. A checksum is simply the result of a function that takes a chunk of data (say a 4KB block) as input and computes a function over said data, producing a small summary of the contents of the data (say 4 or 8 bytes). This summary is referred to as the checksum. The goal of such a computation is to enable a system to detect if data has somehow been corrupted or altered by storing the checksum with the data and then confirming upon later access that the data's current checksum matches the original storage value.

**TIP: THERE'S NO FREE LUNCH**

There's No Such Thing As A Free Lunch, or TNSTAAFL for short, is an old American idiom that implies that when you are seemingly getting something for free, in actuality you are likely paying some cost for it. It comes from the old days when diners would advertise a free lunch for customers, hoping to draw them in; only when you went in, did you realize that to acquire the "free" lunch, you had to purchase one or more alcoholic beverages. Of course, this may not actually be a problem, particularly if you are an aspiring alcoholic (or typical undergraduate student).

## Common Checksum Functions

A number of different functions are used to compute checksums, and vary in strength (i.e., how good they are at protecting data integrity) and speed (i.e., how quickly can they be computed). A trade-off that is common in systems arises here: usually, the more protection you get, the costlier it is. There is no such thing as a free lunch.

One simple checksum function that some use is based on exclusive or (XOR). With XOR-based checksums, the checksum is computed by XOR'ing each chunk of the data block being checksummed, thus producing a single value that represents the XOR of the entire block.

To make this more concrete, imagine we are computing a 4-byte checksum over a block of 16 bytes (this block is of course too small to really be a disk sector or block, but it will serve for the example). The 16 data bytes, in hex, look like this:

```
365e c4cd ba14 8a92 ecef 2c3a 40be f666
```

If we view them in binary, we get the following:

0011	0110	0101	1110	1100	0100	1100	1101
1011	1010	0001	0100	1000	1010	1001	0010
1110	1100	1110	1111	0010	1100	0011	1010
0100	0000	1011	1110	1111	0110	0110	0110

Because we've lined up the data in groups of 4 bytes per row, it is easy to see what the resulting checksum will be: perform an XOR over each column to get the final checksum value:

```
0010 0000 0001 1011      1001 0100 0000 0011
```

The result, in hex, is 0x201b9403.

XOR is a reasonable checksum but has its limitations. If, for example, two bits in the same position within each checksummed unit change, the checksum will not detect the corruption. For this reason, people have investigated other checksum functions.

Another basic checksum function is addition. This approach has the advantage of being fast; computing it just requires performing 2's-complement addition over each chunk of the data, ignoring overflow. It can detect many changes in data, but is not good if the data, for example, is shifted.

A slightly more complex algorithm is known as the **Fletcher checksum**, named (as you might guess) for the inventor, John G. Fletcher [F82]. It is quite simple to compute and involves the computation of two check bytes,  $s_1$  and  $s_2$ . Specifically, assume a block  $D$  consists of bytes  $d_1 \dots d_n$ ;  $s_1$  is defined as follows:  $s_1 = s_1 + d_i \bmod 255$  (computed over all  $d_i$ );  $s_2$  in turn is:  $s_2 = s_2 + s_1 \bmod 255$  (again over all  $d_i$ ) [F04]. The Fletcher checksum is known to be almost as strong as the CRC (described next), detecting all single-bit errors, all double-bit errors, and a large percentage of burst errors [F04].

One final commonly-used checksum is known as a **cyclic redundancy check (CRC)**. Assume you wish to compute the checksum over a data block  $D$ . All you do is treat  $D$  as if it is a large binary number (it is just a string of bits after all) and divide it by an agreed upon value ( $k$ ). The remainder of this division is the value of the CRC. As it turns out, one can implement this binary modulo operation rather efficiently, and hence the popularity of the CRC in networking as well. See elsewhere for more details [M13].

Whatever the method used, it should be obvious that there is no perfect checksum: it is possible two data blocks with non-identical contents will have identical checksums, something referred to as a **collision**. This fact should be intuitive: after all, computing a checksum is taking something large (e.g., 4KB) and producing a summary that is much smaller (e.g., 4 or 8 bytes). In choosing a good checksum function, we are thus trying to find one that minimizes the chance of collisions while remaining easy to compute.

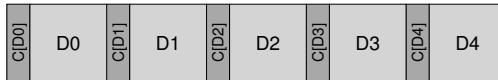
## Checksum Layout

Now that you understand a bit about how to compute a checksum, let's next analyze how to use checksums in a storage system. The first question we must address is the layout of the checksum, i.e., how should checksums be stored on disk?

The most basic approach simply stores a checksum with each disk sector (or block). Given a data block  $D$ , let us call the checksum over that data  $C(D)$ . Thus, without checksums, the disk layout looks like this:

D0	D1	D2	D3	D4	D5	D6
----	----	----	----	----	----	----

With checksums, the layout adds a single checksum for every block:



Because checksums are usually small (e.g., 8 bytes), and disks only can write in sector-sized chunks (512 bytes) or multiples thereof, one problem that arises is how to achieve the above layout. One solution employed by drive manufacturers is to format the drive with 520-byte sectors; an extra 8 bytes per sector can be used to store the checksum.

In disks that don't have such functionality, the file system must figure out a way to store the checksums packed into 512-byte blocks. One such possibility is as follows:



In this scheme, the  $n$  checksums are stored together in a sector, followed by  $n$  data blocks, followed by another checksum sector for the next  $n$  blocks, and so forth. This scheme has the benefit of working on all disks, but can be less efficient; if the file system, for example, wants to overwrite block  $D1$ , it has to read in the checksum sector containing  $C(D1)$ , update  $C(D1)$  in it, and then write out the checksum sector as well as the new data block  $D1$  (thus, one read and two writes). The earlier approach (of one checksum per sector) just performs a single write.

#### 44.4 Using Checksums

With a checksum layout decided upon, we can now proceed to actually understand how to *use* the checksums. When reading a block  $D$ , the client (i.e., file system or storage controller) also reads its checksum from disk  $C_s(D)$ , which we call the **stored checksum** (hence the subscript  $C_s$ ). The client then *computes* the checksum over the retrieved block  $D$ , which we call the **computed checksum**  $C_c(D)$ . At this point, the client compares the stored and computed checksums; if they are equal (i.e.,  $C_s(D) == C_c(D)$ ), the data has likely not been corrupted, and thus can be safely returned to the user. If they do *not* match (i.e.,  $C_s(D) != C_c(D)$ ), this implies the data has changed since the time it was stored (since the stored checksum reflects the value of the data at that time). In this case, we have a corruption, which our checksum has helped us to detect.

Given a corruption, the natural question is what should we do about it? If the storage system has a redundant copy, the answer is easy: try to use it instead. If the storage system has no such copy, the likely answer is to return an error. In either case, realize that corruption detection is not a

magic bullet; if there is no other way to get the non-corrupted data, you are simply out of luck.

#### 44.5 A New Problem: Misdirected Writes

The basic scheme described above works well in the general case of corrupted blocks. However, modern disks have a couple of unusual failure modes that require different solutions.

The first failure mode of interest is called a **misdirected write**. This arises in disk and RAID controllers which write the data to disk correctly, except in the *wrong* location. In a single-disk system, this means that the disk wrote block  $D_x$  not to address  $x$  (as desired) but rather to address  $y$  (thus “corrupting”  $D_y$ ); in addition, within a multi-disk system, the controller may also write  $D_{i,x}$  not to address  $x$  of disk  $i$  but rather to some other disk  $j$ . Thus our question:

**CRUX: HOW TO HANDLE MISDIRECTED WRITES**

How should a storage system or disk controller detect misdirected writes? What additional features are required from the checksum?

The answer, not surprisingly, is simple: add a little more information to each checksum. In this case, adding a **physical identifier (physical ID)** is quite helpful. For example, if the stored information now contains the checksum  $C(D)$  as well as the disk and sector number of the block, it is easy for the client to determine whether the correct information resides within the block. Specifically, if the client is reading block 4 on disk 10 ( $D_{10,4}$ ), the stored information should include that disk number and sector offset, as shown below. If the information does not match, a misdirected write has taken place, and a corruption is now detected. Here is an example of what this added information would look like on a two-disk system. Note that this figure, like the others before it, is not to scale, as the checksums are usually small (e.g., 8 bytes) whereas the blocks are much larger (e.g., 4 KB or bigger):



You can see from the on-disk format that there is now a fair amount of redundancy on disk: for each block, the disk number is repeated within each block, and the offset of the block in question is also kept next to the

block itself. The presence of redundant information should be no surprise, though; redundancy is the key to error detection (in this case) and recovery (in others). A little extra information, while not strictly needed with perfect disks, can go a long ways in helping detect problematic situations should they arise.

#### 44.6 One Last Problem: Lost Writes

Unfortunately, misdirected writes are not the last problem we will address. Specifically, some modern storage devices also have an issue known as a **lost write**, which occurs when the device informs the upper layer that a write has completed but in fact it never is persisted; thus, what remains is left is the old contents of the block rather than the updated new contents.

The obvious question here is: do any of our checksumming strategies from above (e.g., basic checksums, or physical identity) help to detect lost writes? Unfortunately, the answer is no: the old block likely has a matching checksum, and the physical ID used above (disk number and block offset) will also be correct. Thus our final problem:

##### CRUX: HOW TO HANDLE LOST WRITES

How should a storage system or disk controller detect lost writes? What additional features are required from the checksum?

There are a number of possible solutions that can help [K+08]. One classic approach [BS04] is to perform a **write verify** or **read-after-write**; by immediately reading back the data after a write, a system can ensure that the data indeed reached the disk surface. This approach, however, is quite slow, doubling the number of I/Os needed to complete a write.

Some systems add a checksum elsewhere in the system to detect lost writes. For example, Sun's **Zettabyte File System (ZFS)** includes a checksum in each file system inode and indirect block for every block included within a file. Thus, even if the write to a data block itself is lost, the checksum within the inode will not match the old data. Only if the writes to both the inode and the data are lost simultaneously will such a scheme fail, an unlikely (but unfortunately, possible!) situation.

#### 44.7 Scrubbing

Given all of this discussion, you might be wondering: when do these checksums actually get checked? Of course, some amount of checking occurs when data is accessed by applications, but most data is rarely accessed, and thus would remain unchecked. Unchecked data is problematic for a reliable storage system, as bit rot could eventually affect all copies of a particular piece of data.

To remedy this problem, many systems utilize **disk scrubbing** of various forms [K+08]. By periodically reading through *every* block of the system, and checking whether checksums are still valid, the disk system can reduce the chances that all copies of a certain data item become corrupted. Typical systems schedule scans on a nightly or weekly basis.

## 44.8 Overheads Of Checksumming

Before closing, we now discuss some of the overheads of using checksums for data protection. There are two distinct kinds of overheads, as is common in computer systems: space and time.

Space overheads come in two forms. The first is on the disk (or other storage medium) itself; each stored checksum takes up room on the disk, which can no longer be used for user data. A typical ratio might be an 8-byte checksum per 4 KB data block, for a 0.19% on-disk space overhead.

The second type of space overhead comes in the memory of the system. When accessing data, there must now be room in memory for the checksums as well as the data itself. However, if the system simply checks the checksum and then discards it once done, this overhead is short-lived and not much of a concern. Only if checksums are kept in memory (for an added level of protection against memory corruption [Z+13]) will this small overhead be observable.

While space overheads are small, the time overheads induced by checksumming can be quite noticeable. Minimally, the CPU must compute the checksum over each block, both when the data is stored (to determine the value of the stored checksum) as well as when it is accessed (to compute the checksum again and compare it against the stored checksum). One approach to reducing CPU overheads, employed by many systems that use checksums (including network stacks), is to combine data copying and checksumming into one streamlined activity; because the copy is needed anyhow (e.g., to copy the data from the kernel page cache into a user buffer), combined copying/checksumming can be quite effective.

Beyond CPU overheads, some checksumming schemes can induce extra I/O overheads, particularly when checksums are stored distinctly from the data (thus requiring extra I/Os to access them), and for any extra I/O needed for background scrubbing. The former can be reduced by design; the latter can be tuned and thus its impact limited, perhaps by controlling when such scrubbing activity takes place. The middle of the night, when most (not all!) productive workers have gone to bed, may be a good time to perform such scrubbing activity and increase the robustness of the storage system.

## 44.9 Summary

We have discussed data protection in modern storage systems, focusing on checksum implementation and usage. Different checksums protect

against different types of faults; as storage devices evolve, new failure modes will undoubtedly arise. Perhaps such change will force the research community and industry to revisit some of these basic approaches, or invent entirely new approaches altogether. Time will tell. Or it won't. Time is funny that way.

## References

- [B+07] "An Analysis of Latent Sector Errors in Disk Drives"  
 Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, Jiri Schindler  
 SIGMETRICS '07, San Diego, California, June 2007  
*The first paper to study latent sector errors in detail. As described in the next citation [B+08], a collaboration between Wisconsin and NetApp. The paper also won the Kenneth C. Sevcik Outstanding Student Paper award; Sevcik was a terrific researcher and wonderful guy who passed away too soon. To show the authors it was possible to move from the U.S. to Canada and love it, he once sang us the Canadian national anthem, standing up in the middle of a restaurant to do so.*
- [B+08] "An Analysis of Data Corruption in the Storage Stack"  
 Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder,  
 Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau  
 FAST '08, San Jose, CA, February 2008  
*The first paper to truly study disk corruption in great detail, focusing on how often such corruption occurs over three years for over 1.5 million drives. Lakshmi did this work while a graduate student at Wisconsin under our supervision, but also in collaboration with his colleagues at NetApp where he was an intern for multiple summers. A great example of how working with industry can make for much more interesting and relevant research.*
- [BS04] "Commercial Fault Tolerance: A Tale of Two Systems"  
 Wendy Bartlett, Lisa Spainhower  
 IEEE Transactions on Dependable and Secure Computing, Vol. 1, No. 1, January 2004  
*This classic in building fault tolerant systems is an excellent overview of the state of the art from both IBM and Tandem. Another must read for those interested in the area.*
- [C+04] "Row-Diagonal Parity for Double Disk Failure Correction"  
 P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, S. Sankar  
 FAST '04, San Jose, CA, February 2004  
*An early paper on how extra redundancy helps to solve the combined full-disk-failure/partial-disk-failure problem. Also a nice example of how to mix more theoretical work with practical.*
- [F04] "Checksums and Error Control"  
 Peter M. Fenwick  
 Copy Available: <http://www.ostep.org/Citations/checksums-03.pdf>  
*A great simple tutorial on checksums, available to you for the amazing cost of free.*
- [F82] "An Arithmetic Checksum for Serial Transmissions"  
 John G. Fletcher  
 IEEE Transactions on Communication, Vol. 30, No. 1, January 1982  
*Fletcher's original work on his eponymous checksum. Of course, he didn't call it the Fletcher checksum, rather he just didn't call it anything, and thus it became natural to name it after the inventor. So don't blame old Fletcher for this seeming act of braggadocio. This anecdote might remind you of Rubik and his cube; Rubik never called it "Rubik's cube"; rather, he just called it "my cube."*
- [HLM94] "File System Design for an NFS File Server Appliance"  
 Dave Hitz, James Lau, Michael Malcolm  
 USENIX Spring '94  
*The pioneering paper that describes the ideas and product at the heart of NetApp's core. Based on this system, NetApp has grown into a multi-billion dollar storage company. If you're interested in learning more about its founding, read Hitz's autobiography "How to Castrate a Bull: Unexpected Lessons on Risk, Growth, and Success in Business" (which is the actual title, no joking). And you thought you could avoid bull castration by going into Computer Science.*

**[K+08] "Parity Lost and Parity Regained"**

Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

FAST '08, San Jose, CA, February 2008

*This work of ours, joint with colleagues at NetApp, explores how different checksum schemes work (or don't work) in protecting data. We reveal a number of interesting flaws in current protection strategies, some of which have led to fixes in commercial products.*

**[M13] "Cyclic Redundancy Checks"**

Author Unknown

Available: <http://www.mathpages.com/home/kmath458.htm>

*Not sure who wrote this, but a super clear and concise description of CRCs is available here. The internet is full of information, as it turns out.*

**[P+05] "IRON File Systems"**

Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

SOSP '05, Brighton, England, October 2005

*Our paper on how disks have partial failure modes, which includes a detailed study of how file systems such as Linux ext3 and Windows NTFS react to such failures. As it turns out, rather poorly! We found numerous bugs, design flaws, and other oddities in this work. Some of this has fed back into the Linux community, thus helping to yield a new more robust group of file systems to store your data.*

**[RO91] "Design and Implementation of the Log-structured File System"**

Mendel Rosenblum and John Ousterhout

SOSP '91, Pacific Grove, CA, October 1991

*Another reference to this ground-breaking paper on how to improve write performance in file systems.*

**[S90] "Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial"**

Fred B. Schneider

ACM Surveys, Vol. 22, No. 4, December 1990

*This classic paper talks generally about how to build fault tolerant services, and includes many basic definitions of terms. A must read for those building distributed systems.*

**[Z+13] "Zettabyte Reliability with Flexible End-to-end Data Integrity"**

Yupu Zhang, Daniel S. Myers, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

MSST '13, Long Beach, California, May 2013

*Our own work on adding data protection to the page cache of a system, which protects against memory corruption as well as on-disk corruption.*

## Summary Dialogue on Persistence

**Student:** Wow, file systems seem interesting(!), and yet complicated.

**Professor:** That's why me and my spouse do our research in this space.

**Student:** Hold on. Are you one of the professors who wrote this book? I thought we were both just fake constructs, used to summarize some main points, and perhaps add a little levity in the study of operating systems.

**Professor:** Uh... er... maybe. And none of your business! And who did you think was writing these things? (sighs) Anyhow, let's get on with it: what did you learn?

**Student:** Well, I think I got one of the main points, which is that it is much harder to manage data for a long time (persistently) than it is to manage data that isn't persistent (like the stuff in memory). After all, if your machines crashes, memory contents disappear! But the stuff in the file system needs to live forever.

**Professor:** Well, as my friend Kevin Hultquist used to say, "Forever is a long time"; while he was talking about plastic golf tees, it's especially true for the garbage that is found in most file systems.

**Student:** Well, you know what I mean! For a long time at least. And even simple things, such as updating a persistent storage device, are complicated, because you have to care what happens if you crash. Recovery, something I had never even thought of when we were virtualizing memory, is now a big deal!

**Professor:** Too true. Updates to persistent storage have always been, and remain, a fun and challenging problem.

**Student:** I also learned about cool things like disk scheduling, and about data protection techniques like RAID and even checksums. That stuff is cool.

**Professor:** I like those topics too. Though, if you really get into it, they can get a little mathematical. Check out some the latest on erasure codes if you want your brain to hurt.

**Student:** I'll get right on that.

**Professor:** (frowns) I think you're being sarcastic. Well, what else did you like?

**Student:** And I also liked all the thought that has gone into building technology-aware systems, like FFS and LFS. Neat stuff! Being disk aware seems cool. But will it matter anymore, with Flash and all the newest, latest technologies?

**Professor:** Good question! And a reminder to get working on that Flash chapter... (scribbles note down to self) ... But yes, even with Flash, all of this stuff is still relevant, amazingly. For example, Flash Translation Layers (FTLs) use log-structuring internally, to improve performance and reliability of Flash-based SSDs. And thinking about locality is always useful. So while the technology may be changing, many of the ideas we have studied will continue to be useful, for a while at least.

**Student:** That's good. I just spent all this time learning it, and I didn't want it to all be for no reason!

**Professor:** Professors wouldn't do that to you, would they?

# Flash-based SSDs

After decades of hard-disk drive dominance, a new form of persistent storage device has recently gained significance in the world. Generically referred to as **solid-state storage**, such devices have no mechanical or moving parts like hard drives; rather, they are simply built out of transistors, much like memory and processors. However, unlike typical random-access memory (e.g., DRAM), such a **solid-state storage device** (a.k.a., an **SSD**) retains information despite power loss, and thus is an ideal candidate for use in persistent storage of data.

The technology we'll focus on is known as **flash** (more specifically, **NAND-based flash**), which was created by Fujio Masuoka in the 1980s [M+14]. Flash, as we'll see, has some unique properties. For example, to write to a given chunk of it (i.e., a **flash page**), you first have to erase a bigger chunk (i.e., a **flash block**), which can be quite expensive. In addition, writing too often to a page will cause it to **wear out**. These two properties make construction of a flash-based SSD an interesting challenge:

## CRUX: HOW TO BUILD A FLASH-BASED SSD

How can we build a flash-based SSD? How can we handle the expensive nature of erasing? How can we build a device that lasts a long time, given that repeated overwrite will wear the device out? Will the march of progress in technology ever cease? Or cease to amaze?

## I.1 Storing a Single Bit

Flash chips are designed to store one or more bits in a single transistor; the level of charge trapped within the transistor is mapped to a binary value. In a **single-level cell (SLC)** flash, only a single bit is stored within a transistor (i.e., 1 or 0); with a **multi-level cell (MLC)** flash, two bits are encoded into different levels of charge, e.g., 00, 01, 10, and 11 are represented by low, somewhat low, somewhat high, and high levels. There is even **triple-level cell (TLC)** flash, which encodes 3 bits per cell. Overall, SLC chips achieve higher performance and are more expensive.

**TIP: BE CAREFUL WITH TERMINOLOGY**

You may have noticed that some terms we have used many times before (blocks, pages) are being used within the context of a flash, but in slightly different ways than before. New terms are not created to make your life harder (although they may be doing just that), but arise because there is no central authority where terminology decisions are made. What is a block to you may be a page to someone else, and vice versa, depending on the context. Your job is simple: to know the appropriate terms within each domain, and use them such that people well-versed in the discipline can understand what you are talking about. It's one of those times where the only solution is simple but sometimes painful: use your memory.

Of course, there are many details as to exactly how such bit-level storage operates, down at the level of device physics. While beyond the scope of this book, you can read more about it on your own [J10].

## I.2 From Bits to Banks/Planes

As they say in ancient Greece, storing a single bit (or a few) does not a storage system make. Hence, flash chips are organized into **banks** or **planes** which consist of a large number of cells.

A bank is accessed in two different sized units: **blocks** (sometimes called **erase blocks**), which are typically of size 128 KB or 256 KB, and **pages**, which are a few KB in size (e.g., 4KB). Within each bank there are a large number of blocks; within each block, there are a large number of pages. When thinking about flash, you must remember this new terminology, which is different than the blocks we refer to in disks and RAIDs and the pages we refer to in virtual memory.

Figure I.1 shows an example of a flash plane with blocks and pages; there are three blocks, each containing four pages, in this simple example. We'll see below why we distinguish between blocks and pages; it turns out this distinction is critical for flash operations such as reading and writing, and even more so for the overall performance of the device. The most important (and weird) thing you will learn is that to write to a page within a block, you first have to erase the entire block; this tricky detail makes building a flash-based SSD an interesting and worthwhile challenge, and the subject of the second-half of the chapter.

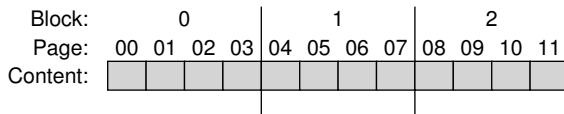


Figure I.1: A Simple Flash Chip: Pages Within Blocks

### I.3 Basic Flash Operations

Given this flash organization, there are three low-level operations that a flash chip supports. The **read** command is used to read a page from the flash; **erase** and **program** are used in tandem to write. The details:

- **Read (a page):** A client of the flash chip can read any page (e.g., 2KB or 4KB), simply by specifying the read command and appropriate page number to the device. This operation is typically quite fast, 10s of microseconds or so, regardless of location on the device, and (more or less) regardless of the location of the previous request (quite unlike a disk). Being able to access any location uniformly quickly means the device is a **random access** device.
- **Erase (a block):** Before writing to a *page* within a flash, the nature of the device requires that you first **erase** the entire *block* the page lies within. Erase, importantly, destroys the contents of the block (by setting each bit to the value 1); therefore, you must be sure that any data you care about in the block has been copied elsewhere (to memory, or perhaps to another flash block) *before* executing the erase. The erase command is quite expensive, taking a few milliseconds to complete. Once finished, the entire block is reset and each page is ready to be programmed.
- **Program (a page):** Once a block has been erased, the program command can be used to change some of the 1's within a page to 0's, and write the desired contents of a page to the flash. Programming a page is less expensive than erasing a block, but more costly than reading a page, usually taking around 100s of microseconds on modern flash chips.

One way to think about flash chips is that each page has a state associated with it. Pages start in an **INVALID** state. By erasing the block that a page resides within, you set the state of the page (and all pages within that block) to **ERASED**, which resets the content of each page in the block but also (importantly) makes them programmable. When you program a page, its state changes to **VALID**, meaning its contents have been set and can be read. Reads do not affect these states (although you should only read from pages that have been programmed). Once a page has been programmed, the only way to change its contents is to erase the entire block within which the page resides. Here is an example of states transition after various erase and program operations within a 4-page block:

		<i>Initial: pages in block are invalid (1)</i>
Erase()	→	<b>EEEE</b> <i>State of pages in block set to erased (E)</i>
Program(0)	→	<b>VEEE</b> <i>Program page 0; state set to valid (V)</i>
Program(0)	→	<b>error</b> <i>Cannot re-program page after programming</i>
Program(1)	→	<b>VVEE</b> <i>Program page 1</i>
Erase()	→	<b>EEEE</b> <i>Contents erased; all pages programmable</i>

## A Detailed Example

Because the process of writing (i.e., erasing and programming) is so unusual, let's go through a detailed example to make sure it makes sense. In this example, imagine we have the following four 8-bit pages, within a 4-page block (both unrealistically small sizes, but useful within this example); each page is **VALID** as each has been previously programmed.

Page 0	Page 1	Page 2	Page 3
00011000	11001110	00000001	00111111
VALID	VALID	VALID	VALID

Now say we wish to write to page 0, filling it with new contents. To write any page, we must first erase the entire block. Let's assume we do so, thus leaving the block in this state:

Page 0	Page 1	Page 2	Page 3
11111111	11111111	11111111	11111111
ERASED	ERASED	ERASED	ERASED

Good news! We could now go ahead and program page 0, for example with the contents 00000011, overwriting the old page 0 (contents 00011000) as desired. After doing so, our block looks like this:

Page 0	Page 1	Page 2	Page 3
00000011	11111111	11111111	11111111
VALID	ERASED	ERASED	ERASED

And now the bad news: the previous contents of pages 1, 2, and 3 are all gone! Thus, before overwriting any page *within* a block, we must first move any data we care about to another location (e.g., memory, or elsewhere on the flash). The nature of erase will have a strong impact on how we design flash-based SSDs, as we'll soon learn about.

## Summary

To summarize, reading a page is easy: just read the page. Flash chips do this quite well, and quickly; in terms of performance, they offer the potential to greatly exceed the random read performance of modern disk drives, which are slow due to mechanical seek and rotation costs.

Writing a page is trickier; the entire block must first be erased (taking care to first move any data we care about to another location), and then the desired page programmed. Not only is this expensive, but frequent repetitions of this program/erase cycle can lead to the biggest reliability problem flash chips have: **wear out**. When designing a storage system with flash, the performance and reliability of writing is a central focus. We'll soon learn more about how modern SSDs attack these issues, delivering excellent performance and reliability despite these limitations.

Device	Read ( $\mu$ s)	Program ( $\mu$ s)	Erase ( $\mu$ s)
SLC	25	200-300	1500-2000
MLC	50	600-900	~3000
TLC	~75	~900-1350	~4500

Figure I.2: Raw Flash Performance Characteristics

## I.4 Flash Performance And Reliability

Because we're interested in building a storage device out of raw flash chips, it is worthwhile to understand their basic performance characteristics. Figure I.2 presents a rough summary of some numbers found in the popular press [V12]. Therein, the author presents the basic operation latency of reads, programs, and erases across SLC, MLC, and TLC flash, which store 1, 2, and 3 bits of information per cell, respectively.

As we can see from the table, read latencies are quite good, taking just 10s of microseconds to complete. Program latency is higher and more variable, as low as 200 microseconds for SLC, but higher as you pack more bits into each cell; to get good write performance, you will have to make use of multiple flash chips in parallel. Finally, erases are quite expensive, taking a few milliseconds typically. Dealing with this cost is central to modern flash storage design.

Let's now consider reliability of flash chips. Unlike mechanical disks, which can fail for a wide variety of reasons (including the gruesome and quite physical **head crash**, where the drive head actually makes contact with the recording surface), flash chips are pure silicon and in that sense have fewer reliability issues to worry about. The primary concern is **wear out**; when a flash block is erased and programmed, it slowly accrues a little bit of extra charge. Over time, as that extra charge builds up, it becomes increasingly difficult to differentiate between a 0 and a 1. At the point where it becomes impossible, the block becomes unusable.

The typical lifetime of a block is currently not well known. Manufacturers rate MLC-based blocks as having a 10,000 P/E (Program/Erase) cycle lifetime; that is, each block can be erased and programmed 10,000 times before failing. SLC-based chips, because they store only a single bit per transistor, are rated with a longer lifetime, usually 100,000 P/E cycles. However, recent research has shown that lifetimes are much longer than expected [BD10].

One other reliability problem within flash chips is known as **disturbance**. When accessing a particular page within a flash, it is possible that some bits get flipped in neighboring pages; such bit flips are known as **read disturbs** or **program disturbs**, depending on whether the page is being read or programmed, respectively.

**TIP: THE IMPORTANCE OF BACKWARDS COMPATIBILITY**

Backwards compatibility is always a concern in layered systems. By defining a stable interface between two systems, one enables innovation on each side of the interface while ensuring continued interoperability. Such an approach has been quite successful in many domains: operating systems have relatively stable APIs for applications, disks provide the same block-based interface to file systems, and each layer in the IP networking stack provides a fixed unchanging interface to the layer above. Not surprisingly, there can be a downside to such rigidity, as interfaces defined in one generation may not be appropriate in the next. In some cases, it may be useful to think about redesigning the entire system entirely. An excellent example is found in the Sun ZFS file system [B07]; by reconsidering the interaction of file systems and RAID, the creators of ZFS envisioned (and then realized) a more effective integrated whole.

## I.5 From Raw Flash to Flash-Based SSDs

Given our basic understanding of flash chips, we now face our next task: how to turn a basic set of flash chips into something that looks like a typical storage device. The standard storage interface is a simple block-based one, where blocks (sectors) of size 512 bytes (or larger) can be read or written, given a block address. The task of the flash-based SSD is to provide that standard block interface atop the raw flash chips inside it.

Internally, an SSD consists of some number of flash chips (for persistent storage). An SSD also contains some amount of volatile (i.e., non-persistent) memory (e.g., SRAM); such memory is useful for caching and buffering of data as well as for mapping tables, which we'll learn about below. Finally, an SSD contains control logic to orchestrate device operation. See Agrawal et. al for details [A+08]; a simplified block diagram is seen in Figure I.3 (page 7).

One of the essential functions of this control logic is to satisfy client reads and writes, turning them into internal flash operations as need be. The **flash translation layer**, or **FTL**, provides exactly this functionality. The FTL takes read and write requests on *logical blocks* (that comprise the device interface) and turns them into low-level read, erase, and program commands on the underlying *physical blocks* and *physical pages* (that comprise the actual flash device). The FTL should accomplish this task with the goal of delivering excellent performance and high reliability.

Excellent performance, as we'll see, can be realized through a combination of techniques. One key will be to utilize multiple flash chips in **parallel**; although we won't discuss this technique much further, suffice it to say that all modern SSDs use multiple chips internally to obtain higher performance. Another performance goal will be to reduce **write amplification**, which is defined as the total write traffic (in bytes) issued to the flash chips by the FTL divided by the total write traffic (in bytes) is-

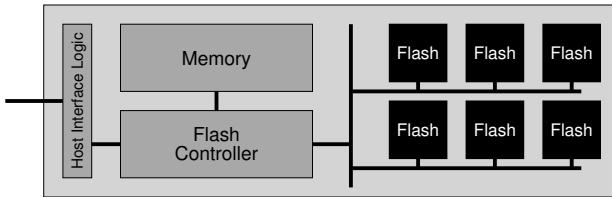


Figure I.3: A Flash-based SSD: Logical Diagram

sued by the client to the SSD. As we'll see below, naive approaches to FTL construction will lead to high write amplification and low performance.

High reliability will be achieved through the combination of a few different approaches. One main concern, as discussed above, is **wear out**. If a single block is erased and programmed too often, it will become unusable; as a result, the FTL should try to spread writes across the blocks of the flash as evenly as possible, ensuring that all of the blocks of the device wear out at roughly the same time; doing so is called **wear leveling** and is an essential part of any modern FTL.

Another reliability concern is program disturbance. To minimize such disturbance, FTLs will commonly program pages within an erased block *in order*, from low page to high page. This sequential-programming approach minimizes disturbance and is widely utilized.

## I.6 FTL Organization: A Bad Approach

The simplest organization of an FTL would be something we call **direct mapped**. In this approach, a read to logical page  $N$  is mapped directly to a read of physical page  $N$ . A write to logical page  $N$  is more complicated; the FTL first has to read in the entire block that page  $N$  is contained within; it then has to erase the block; finally, the FTL programs the old pages as well as the new one.

As you can probably guess, the direct-mapped FTL has many problems, both in terms of performance as well as reliability. The performance problems come on each write: the device has to read in the entire block (costly), erase it (quite costly), and then program it (costly). The end result is severe write amplification (proportional to the number of pages in a block) and as a result, terrible write performance, even slower than typical hard drives with their mechanical seeks and rotational delays.

Even worse is the reliability of this approach. If file system metadata or user file data is repeatedly overwritten, the same block is erased and programmed, over and over, quickly wearing it out and potentially losing data. The direct mapped approach simply gives too much control over wear out to the client workload; if the workload does not spread write load evenly across its logical blocks, the underlying physical blocks containing popular data will quickly wear out. For both reliability and performance reasons, a direct-mapped FTL is a bad idea.

## I.7 A Log-Structured FTL

For these reasons, most FTLs today are **log structured**, an idea useful in both storage devices (as we'll see now) and file systems above them (as we'll see in the chapter on **log-structured file systems**). Upon a write to logical block  $N$ , the device appends the write to the next free spot in the currently-being-written-to block; we call this style of writing **logging**. To allow for subsequent reads of block  $N$ , the device keeps a **mapping table** (in its memory, and persistent, in some form, on the device); this table stores the physical address of each logical block in the system.

Let's go through an example to make sure we understand how the basic log-based approach works. To the client, the device looks like a typical disk, in which it can read and write 512-byte sectors (or groups of sectors). For simplicity, assume that the client is reading or writing 4-KB sized chunks. Let us further assume that the SSD contains some large number of 16-KB sized blocks, each divided into four 4-KB pages; these parameters are unrealistic (flash blocks usually consist of more pages) but will serve our didactic purposes quite well.

Assume the client issues the following sequence of operations:

- Write(100) with contents a1
- Write(101) with contents a2
- Write(2000) with contents b1
- Write(2001) with contents b2

These **logical block addresses** (e.g., 100) are used by the client of the SSD (e.g., a file system) to remember where information is located.

Internally, the device must transform these block writes into the erase and program operations supported by the raw hardware, and somehow record, for each logical block address, which **physical page** of the SSD stores its data. Assume that all blocks of the SSD are currently not valid, and must be erased before any page can be programmed. Here we show the initial state of our SSD, with all pages marked **INVALID** (i):

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]
State:	i	i	i	i	i	i	i	i	i	i	i	i

When the first write is received by the SSD (to logical block 100), the FTL decides to write it to physical block 0, which contains four physical pages: 0, 1, 2, and 3. Because the block is not erased, we cannot write to it yet; the device must first issue an erase command to block 0. Doing so leads to the following state:

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]
State:	E	E	E	E	i	i	i	i	i	i	i	i

Block 0 is now ready to be programmed. Most SSDs will write pages in order (i.e., low to high), reducing reliability problems related to **program disturbance**. The SSD then directs the write of logical block 100 into physical page 0:

But what if the client wants to *read* logical block 100? How can it find where it is? The SSD must transform a read issued to logical block 100 into a read of physical page 0. To accommodate such functionality, when the FTL writes logical block 100 to physical page 0, it records this fact in an **in-memory mapping table**. We will track the state of this mapping table in the diagrams as well:

Now you can see what happens when the client writes to the SSD. The SSD finds a location for the write, usually just picking the next free page; it then programs that page with the block's contents, and records the logical-to-physical mapping in its mapping table. Subsequent reads simply use the table to **translate** the logical block address presented by the client into the physical page number required to read the data.

Let's now examine the rest of the writes in our example write stream: 101, 2000, and 2001. After writing these blocks, the state of the device is:

The log-based approach by its nature improves performance (erases only being required once in a while, and the costly read-modify-write of the direct-mapped approach avoided altogether), and greatly enhances reliability. The FTL can now spread writes across all pages, performing what is called **wear leveling** and increasing the lifetime of the device; we'll discuss wear leveling further below.

#### ASIDE: FTL MAPPING INFORMATION PERSISTENCE

You might be wondering: what happens if the device loses power? Does the in-memory mapping table disappear? Clearly, such information cannot truly be lost, because otherwise the device would not function as a persistent storage device. An SSD must have some means of recovering mapping information.

The simplest thing to do is to record some mapping information with each page, in what is called an **out-of-band (OOB)** area. When the device loses power and is restarted, it must reconstruct its mapping table by scanning the OOB areas and reconstructing the mapping table in memory. This basic approach has its problems; scanning a large SSD to find all necessary mapping information is slow. To overcome this limitation, some higher-end devices use more complex **logging** and **checkpointing** techniques to speed up recovery; we'll learn more about logging later when we discuss file systems.

Unfortunately, this basic approach to log structuring has some downsides. The first is that overwrites of logical blocks lead to something we call **garbage**, i.e., old versions of data around the drive and taking up space. The device has to periodically perform **garbage collection (GC)** to find said blocks and free space for future writes; excessive garbage collection drives up write amplification and lowers performance. The second is high cost of in-memory mapping tables; the larger the device, the more memory such tables need. We now discuss each in turn.

## I.8 Garbage Collection

The first cost of any log-structured approach such as this one is that garbage is created, and therefore **garbage collection** (i.e., dead-block reclamation) must be performed. Let's use our continued example to make sense of this. Recall that logical blocks 100, 101, 2000, and 2001 have been written to the device.

Now, let's assume that blocks 100 and 101 are written to again, with contents *c*1 and *c*2. The writes are written to the next free pages (in this case, physical pages 4 and 5), and the mapping table is updated accordingly. Note that the device must have first erased block 1 to make such programming possible:

Table: 100 → 4 101 → 5 2000 → 2 2001 → 3 Memory

Block:	0				1				2				Flash Chip
Page:	00	01	02	03	04	05	06	07	08	09	10	11	
Content:	a1	a2	b1	b2	c1	c2							
State:	V	V	V	V	V	V	E	E	i	i	i	i	

The problem we have now should be obvious: physical pages 0 and 1, although marked `VALID`, have **garbage** in them, i.e., the old versions of blocks 100 and 101. Because of the log-structured nature of the device, overwrites create garbage blocks, which the device must reclaim to provide free space for new writes to take place.

The process of finding garbage blocks (also called **dead blocks**) and reclaiming them for future use is called **garbage collection**, and it is an important component of any modern SSD. The basic process is simple: find a block that contains one or more garbage pages, read in the live (non-garbage) pages from that block, write out those live pages to the log, and (finally) reclaim the entire block for use in writing.

Let's now illustrate with an example. The device decides it wants to reclaim any dead pages within block 0 above. Block 0 has two dead blocks (pages 0 and 1) and two lives blocks (pages 2 and 3, which contain blocks 2000 and 2001, respectively). To do so, the device will:

- Read live data (pages 2 and 3) from block 0
  - Write live data to end of the log
  - Erase block 0 (freeing it for later usage)

For the garbage collector to function, there must be enough information within each block to enable the SSD to determine whether each page is live or dead. One natural way to achieve this end is to store, at some location within each block, information about which logical blocks are stored within each page. The device can then use the mapping table to determine whether each page within the block holds live data or not.

From our example above (before the garbage collection has taken place), block 0 held logical blocks 100, 101, 2000, 2001. By checking the mapping table (which, before garbage collection, contained  $100 \rightarrow 4$ ,  $101 \rightarrow 5$ ,  $2000 \rightarrow 2$ ,  $2001 \rightarrow 3$ ), the device can readily determine whether each of the pages within the SSD block holds live information. For example, 2000 and 2001 clearly are still pointed to by the map; 100 and 101 are not and therefore are candidates for garbage collection.

When this garbage collection process is complete in our example, the state of the device is:

Table:	100	$\rightarrow$ 4	101	$\rightarrow$ 5	2000	$\rightarrow$ 6	2001	$\rightarrow$ 7	Memory			
Block:	0				1			2		Flash Chip		
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:	E	E	E	E	c1	c2	b1	b2	E	E	E	E
State:	V	V	V	V	i	i	i	i	V	V	V	V

As you can see, garbage collection can be expensive, requiring reading and rewriting of live data. The ideal candidate for reclamation is a block that consists of only dead pages; in this case, the block can immediately be erased and used for new data, without expensive data migration.

To reduce GC costs, some SSDs **overprovision** the device [A+08]; by adding extra flash capacity, cleaning can be delayed and pushed to the **background**, perhaps done at a time when the device is less busy. Adding more capacity also increases internal bandwidth, which can be used for cleaning and thus not harm perceived bandwidth to the client. Many modern drives overprovision in this manner, one key to achieving excellent overall performance.

## I.9 Mapping Table Size

The second cost of log-structuring is the potential for extremely large mapping tables, with one entry for each 4-KB page of the device. With a large 1-TB SSD, for example, a single 4-byte entry per 4-KB page results in 1 GB of memory needed the device, just for these mappings! Thus, this **page-level** FTL scheme is impractical.

### Block-Based Mapping

One approach to reduce the costs of mapping is to only keep a pointer per *block* of the device, instead of per page, reducing the amount of mapping information by a factor of  $\frac{\text{Size}_{\text{block}}}{\text{Size}_{\text{page}}}$ . This **block-level** FTL is akin to having bigger page sizes in a virtual memory system; in that case, you use fewer bits for the VPN and have a larger offset in each virtual address.

Unfortunately, using a block-based mapping inside a log-based FTL does not work very well for performance reasons. The biggest problem arises when a “small write” occurs (i.e., one that is less than the size of a physical block). In this case, the FTL must read a large amount of live data from the old block and copy it into a new one (along with the data from the small write). This data copying increases write amplification greatly and thus decreases performance.

To make this issue more clear, let’s look at an example. Assume the client previously wrote out logical blocks 2000, 2001, 2002, and 2003 (with contents, a, b, c, d), and that they are located within physical block 1 at physical pages 4, 5, 6, and 7. With per-page mappings, the translation table would have to record four mappings for these logical blocks: 2000→4, 2001→5, 2002→6, 2003→7.

If, instead, we use block-level mapping, the FTL only needs to record a single address translation for all of this data. The address mapping, however, is slightly different than our previous examples. Specifically, we think of the logical address space of the device as being chopped into chunks that are the size of the physical blocks within the flash. Thus, the logical block address consists of two portions: a chunk number and an offset. Because we are assuming four logical blocks fit within each physical block, the offset portion of the logical addresses requires 2 bits; the remaining (most significant) bits form the chunk number.

Logical blocks 2000, 2001, 2002, and 2003 all have the same chunk number (500), and have different offsets (0, 1, 2, and 3, respectively). Thus, with a block-level mapping, the FTL records that chunk 500 maps to block 1 (starting at physical page 4), as shown in this diagram:

Table: 500 → 4

Memory												
Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:	[ ]	[ ]	[ ]	[ ]	a	b	c	d	[ ]	[ ]	[ ]	[ ]
State:	i	i	i	i	V	V	V	V	i	i	i	i

Flash Chip

In a block-based FTL, reading is easy. First, the FTL extracts the chunk number from the logical block address presented by the client, by taking the topmost bits out of the address. Then, the FTL looks up the chunk-number to physical-page mapping in the table. Finally, the FTL computes the address of the desired flash page by *adding* the offset from the logical address to the physical address of the block.

For example, if the client issues a read to logical address 2002, the device extracts the logical chunk number (500), looks up the translation in the mapping table (finding 4), and adds the offset from the logical address (2) to the translation (4). The resulting physical-page address (6) is where the data is located; the FTL can then issue the read to that physical address and obtain the desired data (c).

But what if the client writes to logical block 2002 (with contents c')? In this case, the FTL must read in 2000, 2001, and 2003, and then write out all four logical blocks in a new location, updating the mapping table accordingly. Block 1 (where the data used to reside) can then be erased and reused, as shown here.

Table: 500 → 8

Memory												
Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	a	b	c'	d
State:	i	i	i	i	E	E	E	E	V	V	V	V

Flash Chip

As you can see from this example, while block level mappings greatly reduce the amount of memory needed for translations, they cause significant performance problems when writes are smaller than the physical block size of the device; as real physical blocks can be 256KB or larger, such writes are likely to happen quite often. Thus, a better solution is needed. Can you sense that this is the part of the chapter where we tell you what that solution is? Better yet, can you figure it out yourself, before reading on?

## Hybrid Mapping

To enable flexible writing but also reduce mapping costs, many modern FTLs employ a **hybrid mapping** technique. With this approach, the FTL keeps a few blocks erased and directs all writes to them; these are called **log blocks**. Because the FTL wants to be able to write any page to any location within the log block without all the copying required by a pure block-based mapping, it keeps *per-page* mappings for these log blocks.

The FTL thus logically has two types of mapping table in its memory: a small set of per-page mappings in what we'll call the *log table*, and a larger set of per-block mappings in the *data table*. When looking for a particular logical block, the FTL will first consult the log table; if the logical block's location is not found there, the FTL will then consult the data table to find its location and then access the requested data.

The key to the hybrid mapping strategy is keeping the number of log blocks small. To keep the number of log blocks small, the FTL has to periodically examine log blocks (which have a pointer per page) and *switch* them into blocks that can be pointed to by only a single block pointer. This switch is accomplished by one of three main techniques, based on the contents of the block [KK+02].

For example, let's say the FTL had previously written out logical pages 1000, 1001, 1002, and 1003, and placed them in physical block 2 (physical pages 8, 9, 10, 11); assume the contents of the writes to 1000, 1001, 1002, and 1003 are *a*, *b*, *c*, and *d*, respectively.

Log Table:

Data Table: 250 → 8

Memory

Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	
Content:	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	Flash Chip
State:	i	i	i	i	i	i	i	i	V	V	V	V	

Now assume that the client overwrites each of these pages (with data *a'*, *b'*, *c'*, and *d'*), in the exact same order, in one of the currently available log blocks, say physical block 0 (physical pages 0, 1, 2, and 3). In this case, the FTL will have the following state:

Log Table: 1000 → 0 1001 → 1 1002 → 2 1003 → 3

Data Table: 250 → 8

Memory

Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	
Content:	a'	b'	c'	d'	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	Flash Chip
State:	V	V	V	V	i	i	i	i	V	V	V	V	

Because these blocks have been written exactly in the same manner as before, the FTL can perform what is known as a **switch merge**. In this case, the log block (0) now becomes the storage location for pages 0, 1, 2, and 3, and is pointed to by a single block pointer; the old block (2) is now erased and used as a log block. In this best case, all the per-page pointers required replaced by a single block pointer.

Log Table:

Data Table: 250 → 0

Memory

Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash Chip
Content:	a'	b'	c'	d'									
State:	V	V	V	V	i	i	i	i	i	i	i	i	

This switch merge is the best case for a hybrid FTL. Unfortunately, sometimes the FTL is not so lucky. Imagine the case where we have the same initial conditions (logical blocks 0, 1, 2, and 4 stored in physical block 2) but then the client overwrites only logical blocks 0 and 1:

Log Table: 1000 → 0 1001 → 1

Data Table: 250 → 8

Memory

Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash Chip
Content:	a'	b'							a	b	c	d	
State:	V	V	i	i	i	i	i	i	V	V	V	V	

To reunite the other pages of this physical block, and thus be able to refer to them by only a single block pointer, the FTL performs what is called a **partial merge**. In this operation, 2 and 3 are read from block 4, and then appended to the log. The resulting state of the SSD is the same as the switch merge above; however, in this case, the FTL had to perform extra I/O to achieve its goals (in this case, reading logical blocks 2 and 3 from physical pages 18 and 19, and then writing them out to physical pages 22 and 23), thus increasing write amplification.

The final case encountered by the FTL known as a **full merge**, and requires even more work. In this case, the FTL must pull together pages from many other blocks to perform cleaning. For example, imagine that pages 0, 4, 8, and 12 are written to log block A. To switch this log block into a block-mapped page, the FTL must first create a data block containing logical blocks 0, 1, 2, and 3, and thus the FTL must read 1, 2, and 3 from elsewhere and then write out 0, 1, 2, and 3 together. Next, the merge must do the same for logical block 4, finding 5, 6, and 7 and reconciling them into a single data block. The same must be done for logical blocks 8 and 12, and then (finally), the log block A can be freed. Frequent full merges, as is not surprising, can seriously hurt performance [GY+09].

## I.10 Wear Leveling

Finally, a related background activity that modern FTLs must implement is **wear leveling**, as introduced above. The basic idea is simple: because multiple erase/program cycles will wear out a flash block, the FTL should try its best to spread that work across all the blocks of the device evenly. In this manner, all blocks will wear out at roughly the same time, instead of a few “popular” blocks quickly unusable.

The basic log-structuring approach does a good initial job of spreading out write load, and garbage collection helps as well. However, sometimes a block will be filled with long-lived data that does not get over-written; in this case, garbage collection will never reclaim the block, and thus it does not receive its fair share of the write load.

To remedy this problem, the FTL must periodically read all the live data out of such blocks and re-write it elsewhere, thus making the block available for writing again. This process of wear leveling increases the write amplification of the SSD, and thus decreases performance as extra I/O is required to ensure that all blocks wear at roughly the same rate. Many different algorithms exist in the literature [A+08, M+14]; read more if you are interested.

## I.11 SSD Performance And Cost

Before closing, let’s examine the performance and cost of modern SSDs, to better understand how they will likely be used in persistent storage systems. In both cases, we’ll compare to classic hard-disk drives (HDDs), and highlight the biggest differences between the two.

### Performance

Unlike hard disk drives, flash-based SSDs have no mechanical components, and in fact are in many ways more similar to DRAM, in that they are “random access” devices. The biggest difference in performance, as compared to disk drives, is realized when performing random reads and writes; while a typical disk drive can only perform a few hundred random I/Os per second, SSDs can do much better. Here, we use some data from modern SSDs to see just how much better SSDs perform; we’re particularly interested in how well the FTLs hide the performance issues of the raw chips.

Table I.4 shows some performance data for three different SSDs and one top-of-the-line hard drive; the data was taken from a few different online sources [S13, T15]. The left two columns show random I/O performance, and the right two columns sequential; the first three rows show data for three different SSDs (from Samsung, Seagate, and Intel), and the last row shows performance for a **hard disk drive** (or HDD), in this case a Seagate high-end drive.

Device	Random		Sequential	
	Reads (MB/s)	Writes (MB/s)	Reads (MB/s)	Writes (MB/s)
Samsung 840 Pro SSD	103	287	421	384
Seagate 600 SSD	84	252	424	374
Intel SSD 335 SSD	39	222	344	354
Seagate Savvio 15K.3 HDD	2	2	223	223

Figure I.4: SSDs And Hard Drives: Performance Comparison

We can learn a few interesting facts from the table. First, and most dramatic, is the difference in random I/O performance between the SSDs and the lone hard drive. While the SSDs obtain tens or even hundreds of MB/s in random I/Os, this “high performance” hard drive has a peak of just a couple MB/s (in fact, we rounded up to get to 2 MB/s). Second, you can see that in terms of sequential performance, there is much less of a difference; while the SSDs perform better, a hard drive is still a good choice if sequential performance is all you need. Third, you can see that SSD random read performance is not as good as SSD random write performance. The reason for such unexpectedly good random-write performance is due to the log-structured design of many SSDs, which transforms random writes into sequential ones and improves performance. Finally, because SSDs exhibit some performance difference between sequential and random I/Os, many of the techniques we will learn in subsequent chapters about how to build file systems for hard drives are still applicable to SSDs; although the magnitude of difference between sequential and random I/Os is smaller, there is enough of a gap to carefully consider how to design file systems to reduce random I/Os.

### Cost

As we saw above, the performance of SSDs greatly outstrips modern hard drives, even when performing sequential I/O. So why haven’t SSDs completely replaced hard drives as the storage medium of choice? The answer is simple: cost, or more specifically, cost per unit of capacity. Currently [A15], an SSD costs something like \$150 for a 250-GB drive; such an SSD costs 60 cents per GB. A typical hard drive costs roughly \$50 for 1-TB of storage, which means it costs 5 cents per GB. There is still more than a 10× difference in cost between these two storage media.

These performance and cost differences dictate how large-scale storage systems are built. If performance is the main concern, SSDs are a terrific choice, particularly if random read performance is important. If, on the other hand, you are assembling a large data center and wish to store massive amounts of information, the large cost difference will drive you towards hard drives. Of course, a hybrid approach can make sense – some storage systems are being assembled with both SSDs and hard drives, using a smaller number of SSDs for more popular “hot” data and delivering high performance, while storing the rest of the “colder” (less used) data on hard drives to save on cost. As long as the price gap exists, hard drives are here to stay.

## I.12 Summary

Flash-based SSDs are becoming a common presence in laptops, desktops, and servers inside the datacenters that power the world’s economy. Thus, you should probably know something about them, right?

Here’s the bad news: this chapter (like many in this book) is just the first step in understanding the state of the art. Some places to get some more information about the raw technology include research on actual device performance (such as that by Chen et al. [CK+09] and Grupp et al. [GC+09]), issues in FTL design (including works by Agrawal et al. [A+08], Gupta et al. [GY+09], Huang et al. [H+14], Kim et al. [KK+02], Lee et al. [L+07], and Zhang et al. [Z+12]), and even distributed systems comprised of flash (including Gordon [CG+09] and CORFU [B+12]).

Don’t just read academic papers; also read about recent advances in the popular press (e.g., [V12]). Therein you’ll learn more practical (but still useful) information, such as Samsung’s use of both TLC and SLC cells within the same SSD to maximize performance (SLC can buffer writes quickly) as well as capacity (TLC can store more bits per cell). And this is, as they say, just the tip of the iceberg. Dive in and learn more about this “iceberg” of research on your own, perhaps starting with Ma et al.’s excellent (and recent) survey [M+14]. Be careful though; icebergs can sink even the mightiest of ships [W15].

## References

- [A+08] "Design Tradeoffs for SSD Performance"  
 N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy  
 USENIX '08, San Diego California, June 2008  
*An excellent overview of what goes into SSD design.*
- [A15] "Amazon Pricing Study"  
 Remzi Arpaci-Dusseau  
 February, 2015  
*This is not an actual paper, but rather one of the authors going to Amazon and looking at current prices of hard drives and SSDs. You too can repeat this study, and see what the costs are today. Do it!*
- [B+12] "CORFU: A Shared Log Design for Flash Clusters"  
 M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, J. D. Davis  
 NSDI '12, San Jose, California, April 2012  
*A new way to think about designing a high-performance replicated log for clusters using Flash.*
- [BD10] "Write Endurance in Flash Drives: Measurements and Analysis"  
 Simona Boboila, Peter Desnoyers  
 FAST '10, San Jose, California, February 2010  
*A cool paper that reverse engineers flash-device lifetimes. Endurance sometimes far exceeds manufacturer predictions, by up to 100×.*
- [B07] "ZFS: The Last Word in File Systems"  
 Jeff Bonwick and Bill Moore  
 Available: <http://opensolaris.org/os/community/zfs/docs/zfs.last.pdf>  
*Was this the last word in file systems? No, but maybe it's close.*
- [CG+09] "Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications"  
 Adrian M. Caulfield, Laura M. Grupp, Steven Swanson  
 ASPLOS '09, Washington, D.C., March 2009  
*Early research on assembling flash into larger-scale clusters; definitely worth a read.*
- [CK+09] "Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives"  
 Feng Chen, David A. Koufaty, and Xiaodong Zhang  
 SIGMETRICS/Performance '09, Seattle, Washington, June 2009  
*An excellent overview of SSD performance problems circa 2009 (though now a little dated).*
- [G14] "The SSD Endurance Experiment"  
 Geoff Gasior  
 The Tech Report, September 19, 2014  
 Available: <http://techreport.com/review/27062>  
*A nice set of simple experiments measuring performance of SSDs over time. There are many other similar studies; use google to find more.*
- [GC+09] "Characterizing Flash Memory: Anomalies, Observations, and Applications"  
 L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, J. K. Wolf  
 IEEE MICRO '09, New York, New York, December 2009  
*Another excellent characterization of flash performance.*
- [GY+09] "DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings"  
 Aayush Gupta, Youngjae Kim, Bhuvan Urgaonkar  
 ASPLOS '09, Washington, D.C., March 2009  
*This paper gives an excellent overview of different strategies for cleaning within hybrid SSDs as well as a new scheme which saves mapping table space and improves performance under many workloads.*

[H+14] "An Aggressive Wear-out Flash Block Management Scheme To Alleviate SSD Performance Degradation"

Ping Huang, Guanying Wu, Xubin He, Weijun Xiao  
EuroSys '14, 2014

*Recent work showing how to really get the most out of worn-out flash blocks; neat!*

[J10] "Failure Mechanisms and Models for Semiconductor Devices"

Report JEP122F, November 2010

Available: <http://www.jedec.org/sites/default/files/docs/JEP122F.pdf>

*A highly detailed discussion of what is going on at the device level and how such devices fail. Only for those not faint of heart. Or physicists. Or both.*

[KK+02] "A Space-Efficient Flash Translation Layer For Compact Flash Systems"

Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho

IEEE Transactions on Consumer Electronics, Volume 48, Number 2, May 2002

*One of the earliest proposals to suggest hybrid mappings.*

[L+07] "A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation"

Sang-won Lee, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, Ha-Joo Song

ACM Transactions on Embedded Computing Systems, Volume 6, Number 3, July 2007

*A terrific paper about how to build hybrid log/block mappings.*

[M+14] "A Survey of Address Translation Technologies for Flash Memories"

Dongzhe Ma, Jianhua Feng, Guoliang Li

ACM Computing Surveys, Volume 46, Number 3, January 2014

*Probably the best recent survey of flash and related technologies.*

[S13] "The Seagate 600 and 600 Pro SSD Review"

Anand Lal Shimpi

AnandTech, May 7, 2013

Available: <http://www.anandtech.com/show/6935/seagate-600-ssd-review>

*One of many SSD performance measurements available on the internet. Haven't heard of the internet? No problem. Just go to your web browser and type "internet" into the search tool. You'll be amazed at what you can learn.*

[T15] "Performance Charts Hard Drives"

Tom's Hardware, January 2015

Available: <http://www.tomshardware.com/charts/enterprise-hdd-charts/>

*Yet another site with performance data, this time focusing on hard drives.*

[V12] "Understanding TLC Flash"

Kristian Vatto

AnandTech, September, 2012

Available: <http://www.anandtech.com/show/5067/understanding-tlc-nand>

*A short description about TLC flash and its characteristics.*

[W15] "List of Ships Sunk by Icebergs"

Available: [http://en.wikipedia.org/wiki/List\\_of\\_ships\\_sunk\\_by\\_icebergs](http://en.wikipedia.org/wiki/List_of_ships_sunk_by_icebergs)

*Yes, there is a wikipedia page about ships sunk by icebergs. It is a really boring page and basically everyone knows the only ship the iceberg-sinking-mafia cares about is the Titanic.*

[Z+12] "De-indirection for Flash-based SSDs with Nameless Writes"

Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

FAST '13, San Jose, California, February 2013

*Our research on a new idea to reduce mapping table space; the key is to re-use the pointers in the file system above to store locations of blocks, instead of adding another level of indirection.*

# **Security for Operating Systems**

## **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. 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. It can make your memory readable or writeable by another process, it can starve you of cycles, it can corrupt the messages you send via your network card, it can wipe data off your disk, or do pretty much anything else with the hardware you rely on. You trust the operating system not to do any of those things, but your trust relies on the assumption that the OS contains no flaws that will allow an attacker to make it misbehave.

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 opponent 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. 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 opponents, 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. So 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 we fully trusted every process running on our machine to do anything it wants with any hardware resource and any piece of data on the machine, 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 unwritable does not get altered. If the applications cannot rely on proper implementation of OS abstraction security guarantees, then they cannot use these abstractions to achieve their own security goals. At the minimum, that implies a great deal more work on 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 and a well-defined scheduling policy) cannot be trusted.

### THE CRUX OF THE PROBLEM HOW CAN WE 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?

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 even if you don't personally care at all about security, understanding what your OS does to secure itself is necessary to understand how to get the system to do what you want.

## 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 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' 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
- give any process access to any other process' 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<sup>1</sup>, 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

---

<sup>1</sup> If you suspect your operating system is malicious, it's time to get a new operating system.

completely compromise everything about the machine the system runs on, so preventing them and patching any that are found is vitally important.

## 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* – Keep your secrets. 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 opponent 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.
- *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.

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' 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.

#### 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' 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 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 they are also potential channels for malicious attacks. Finally, the OS does not always have complete control of the hardware . . .

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 very 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<sup>2</sup>.

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

---

<sup>2</sup> Yet another example of the operating system using general mechanisms to implement flexible policies.

system precisely what that policy is. With some important exceptions (like maintaining a process' address space private unless specifically directed otherwise), the operating system merely supplies general mechanisms that can implement many specific policies. Without intelligent design of policies and careful application of the mechanisms, however, what the operating system can do may not be what your operating system will do.

## 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 more 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 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<sup>3</sup>.
4. Open design – Assume your opponent 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.

---

<sup>3</sup> 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.

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.

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 [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 specifics of the language. For a comprehensive treatment on how to evaluate if your system is secure, start with [D+07].

## 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 main 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.

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 chapters. 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.

#### TIP: 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.

## 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 by all systems.

The next question to address is, what mechanisms should our operating system contain 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”  
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.*

[SE13] “Secure Coding in C and C++”  
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”  
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.*

# Authentication for Operating Systems

## 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 you 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.

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, and system calls are made by particular processes. As discussed in the first part of this book, system calls result in a trap from user code into the operating system. The operating system then takes control and performs some service in response to the system call. At the moment the operating system takes control, it is able to determine which process made the call. Associated with that 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 some cases, other information specific to the process is useful for making this decision. For instance, as we discovered when investigating virtualization of memory, per-process data structures ensure that only memory page frames assigned to a process can be accessed by that process. Also, when the operating system grants a process access to some resource, such as a file or a hardware device, an indication of that permission can be attached to the process control block, which will allow the operating system to determine, on later requests to use the resource, whether they should or shouldn't be permitted.

In other cases, however, the operating system needs more information about the process' identity to determine if its request should be granted. Often, some higher-level identity associated with a process is needed to make this decision. Different operating systems

have used different types of identity for this purpose. 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 the power of the concept, 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<sup>1</sup>. 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 misidentify a janitor employee process as an accounting department employee process, you could end up with an empty bank account. (And, since the janitor is fleeing to Rio with the loot, your trash cans won't get emptied.) Or if you fail to identify your company president correctly when he's trying to give an important presentation to investors, you may find yourself out of a job once he determines that you're the one who preventing him from getting the next round of startup capital. 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. Which leads us to the crux of the problem.

### THE CRUX OF THE PROBLEM HOW CAN WE SECURELY IDENTIFY PROCESSES?

For systems that have multiple identities associated with processes, 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 change the identity of a process?

## Attaching Identities to Processes

Where do processes come from? Usually they are created by other processes, though of course there's always that question of where the first process came from. One simple way to attach an identity to a new process, then, is simply to copy the identity of the

---

<sup>1</sup> 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.

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 the 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, what's the point in keeping track of it in the process control block? Since there's only one, you already know the identity. And, of course, if there's only one identity and we're going to use process identity to make security decisions about processes, we're not going to be able to implement any policy that distinguishes the handling of one process versus another.

We essentially have two choices on how to move forward: first, we can arrange that some processes have different identities and use that difference to manage our security policies. Or, second, we can use some other means to attach information to processes that will allow us to implement our policies. The second option immediately raises the question of how do we determine which information to attach to each process, which more or less leads us in a circle. Ultimately, if we want some processes to have different security policy treatment than others, we need, somehow or other, to identify which processes get which treatment. And that leads us back to process identity, in one way or another.

An easy escape from such circularity is simply to accept that there are different identities in the system and security treatment depends on determining which identity each process should have. A simple case is a multi-user system. We could then assign processes identities 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.

OK, so processes have a security-relevant identity, like a user ID. If not all processes belong to the same user ID, we're going to have to do something to set the proper user ID for a new process. In the ordinary case, a user has a process that he 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 him. Since the operating system can fiddle with its own data structures, like the process control block, it can set the new process' ownership to the user who just joined the system. Of course, we don't want users arbitrarily fiddling with the ownership of their processes themselves, but since the OS

controls the process control block, the user can't alter the ownership information. Well, not without the OS's assistance, at least.

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 he 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 sort of building a towering security structure with unforeseeable implications based on the OS making the right decision here, so it's very important. Let's look at our options.

## 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 computer's 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 his attempt to sneak in. Second, if he is an authorized user, we need to determine, which one?

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

1. Authentication based on what you know
2. Authentication based on what you have
3. 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 [p46], 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 taken to an imperial procurator at some time in the 2<sup>nd</sup> century AD [C00]. 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 [J]. This was a form of authentication by what you are: a native speaker of the Gileadites' dialect or a speaker 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.

## 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 his 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 does 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 must 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 him based on him 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<sup>2</sup>.

Actually, though, our system does not 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 to see 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 claims to be the password, hash his claim and compare it to the stored hashed value. If it matches, you believe he 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. There are some extra wrinkles here, but we’ll leave those aside for the moment.

---

<sup>2</sup> “Might” is too weak a word. The first known incident of such stored passwords leaking is from 1962 [MT79], and such leaks happen to this day with depressing regularity, and general much larger scope. [KA16] discusses a 2016 leak of over 100 million passwords stored in plaintext 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, 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.

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, like all other bits. If your password is a single bit long, then an attacker can try guessing "0" and have a 50/50 chance of being right. Even if he's wrong, if he can guess a second time, he 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, he'll sooner or later guess 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 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 guess is called a *dictionary attack*, and it can be highly effective. The dictionary here isn't Websters (or even the OED), 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].

There are other troubling issues for the use of passwords, but many of those are not particular to operating systems, 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, or PIN. That's essentially a password. But you also need to provide further evidence of your identity . . .

## **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 is, 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?

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 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 we have to the system we need to authenticate ourselves to. 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 he was supposed to type 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. 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 it on your dresser bureau this morning? What if it 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 it? 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, he'll still have to guess that before he can pose as you.

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 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 his message into the computer. Thinking about this in theory, this 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 him 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<sup>3</sup>. Keep this lesson in mind. Even if it isn't on the test, it may come in handy some time in your later career.

## **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. Each person has 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

---

<sup>3</sup> 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.

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 around the same extent that the other authentication mechanisms can be made to work. In other words, we can use it, but it is not likely to be perfect. And it has its own set of characteristic problems and challenges, different from those when you authenticate based on what you know or what you have, but not necessarily any easier to deal with.

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 then determining if it belongs to a particular person or not. Spend a moment thinking about what that entails. To give you thinking some solidity, 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.

If you actually did think about it, like we told you to, you should have identified a few challenges here. Let's walk through things step by step, talking about some of those challenges. 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 he's the actual owner. What if it's someone who looks a lot like the right user, but isn't? What if he's wearing a mask? What if he holds up a photo of the right user, instead of showing his own face? What if the lighting is dim, or he's not fully facing the camera? Alternately, what if it is the right user and he's not facing the camera, or the lighting is dim, or he just shaved off his beard?

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. So that "photo" you took is actually a collection of numbers, indicating shadow and light, shades of color, contrasts, and the like. So whoever you took a picture of, you've converted it to zeros and ones. OK, now what? Time to decide if it's the right person's photo or not! How is your program going to do that?

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 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. There are a lot of ways we can do this, but generally they will involve trying to extract 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.

#### ASIDE: SECURITY SNAKE OIL

If you spend much time worrying about security in a production environment, you'll run into a lot of folks who would like you to pay them for their security product, which invariably is described as tremendously improving the security of your environment. Some of these products are excellent and will indeed help you secure your systems. Others, however, are not any use at all. Like Old West medicine show barkers, they're just selling snake oil that won't cure anything.

Biometrics are one area where this phenomena arises. If you listen to the manufacturers, no one ever marketed a biometric authentication system that was less than perfect. Unfortunately, that's not always quite the truth. A good cautionary tale comes from Japanese researchers who obtained copies of 11 commercially available fingerprint readers, went to a hobby store and purchased around \$10 worth of supplies, and tried to deceive the readers. They used the hobby supplies to make gummy fingers, copying fingerprints onto them in various ways. They were able to fool the fingerprint readers at rates between 68% and 100% using their gummy fingers [M+02].

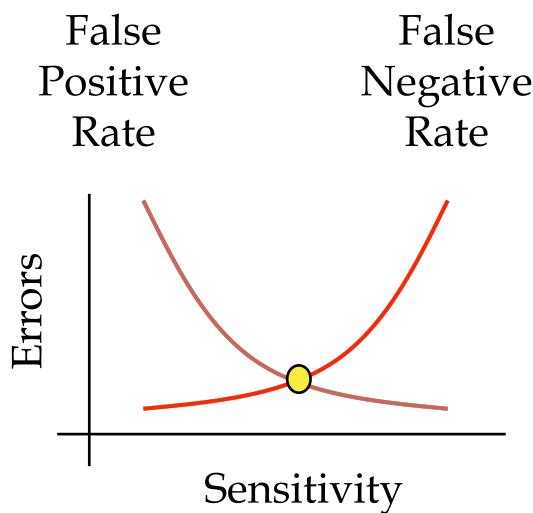
The bigger the claims, the less likely they are to be true. The more "revolutionary" the technology, the more likely it is to be ineffective. To complicate life, that's not always true, but remember that an extraordinary claim requires extraordinary proof [TR78]

Even here, though, an exact match is not too likely. The lighting, for example, might slightly alter the perceived eye color: Paul-Newman blue eyes might become Henry-Fonda blue eyes under different lighting. 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 could go into this issue in a lot more detail, but the important point here is that when we are comparing facial features, or any other biometric, we cannot require perfect matches. Instead, we will look for close 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 the measured versus stored data, we will authenticate a user who

is not who he claims to be. (That's called 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 can't usually minimize both. As the false positive rate goes down, the false negative rate goes up, and vice versa.

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, once in a while requiring the bank manager to provide a scan a second time isn't too bad, while allowing a robber to open the vault with his bogus fake eye would be a disaster. Low false positives might be better here.



**Figure 1. The Crossover Error Rate**

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. Even if the hardware device is available, the convenience of using them for this purpose can be limiting.

#### 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 someone behind that counter, perhaps giving him a bunch of personal information, maybe even giving him some 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 him, he didn't show you an official ID card, he didn't recite the secret DMV mantra that proved he was an initiate of that agency. You believed it because he was standing behind a particular counter, which is the counter DMV employees stand behind. You authenticated him based on where he was.

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. 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.

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 is bits, and anyone can create any bit pattern they want. If an 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 the system and its environment. A good fingerprint reader built in to a smart phone might be pretty good at its job. 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.

## 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 middleman (here, the login) and provide some mechanism whereby the privileged user is permitted to create processes that belong not to him, 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,

```
sudo -u webserver apache2
```

would indicate that the `apache2` program should be started under the identity of `webserver`, rather than under the identity of whoever ran it. This command might require the user running it to provide his own authentication credentials (for extra certainty that it really is the privileged user asking for it, and not some random visitor accessing his 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`.

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. When one of them wished to do something requiring group membership, we would check that she was a member. We can either associate a group membership with each process, or use the process' 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.

## 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 his 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 he has (like a smart card or proof of possession of his smart phone), or something he is (like his 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 in more than one form, such as requiring both a password and a one-time code that was texted to his smart phone.

## References

[C00] Letter of recommendation to Tiberius Claudius Hermeros  
Celer the Architect  
Circa 100 A.D.

*This letter introduced a slave to the imperial procurator, thus providing said curator evidence that the slave was who he claimed to be. You can read it in translation at <http://papyri.info/ddbdp/c.ep.lat;:81>.*

[G13] “Anatomy of a hack: even your 'complicated' password is easy to crack”  
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.*

[J] Judges 12, verses 5-6

The Bible

*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 Judges.*

[KA16] “VK.com Hacked! 100 Million Clear Text Passwords Leaked Online”

Swati Khandelwal

<http://thehackernews.com/2016/06/vk-com-data-breach.html>

*One of many recent reports of stolen passwords stored in plaintext form.*

[MT79] “Password Security: A Case History”

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.*

[M+02] “Impact of Artificial "Gummy" Fingers on Fingerprint Systems”

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.*

[P46] “The Histories”

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”

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.*

# Access Control

## 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 part of this problem 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 solid, 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 something like

```
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?

## 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 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.

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) access 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<sup>1</sup>, 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 hardware. If a process is given access to some virtual device, which is actually backed up by a real physical device controlled by the OS, 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

---

<sup>1</sup> 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.

appears to be theirs alone is actually shared, with the operating system running around behind the scenes to keep the illusion going. 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 abandon 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) 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 way would be to hire a massive intimidating doorman and give him a list of all the approved members. When someone wanted to enter the club, he would prove to the doorman who he was and the doorman would look him up on the list. If it was Linus Torvalds, the doorman would let him in, but he'd keep out the hoi polloi networking folks who had failed to distinguish themselves in the field of 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 architect with no OS chops wanted to get in, he wouldn't have the key and would be stuck outside. Compared to the other approach, we'd save on the salary of the doorman, though we would have to pay for the locks and keys<sup>2</sup>. 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 his 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

---

<sup>2</sup> 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 mask gets past our doorman, or if we aren't careful to determine that it really is Jerome Saltzer before handing a random guy 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.

approach based on the doorman 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.

## Using ACLs for Access Control

What if, in the tradition of old British men's 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' 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 he isn't. If not, no access for him. If he is on the list, we'll typically go a step further to determine if the ACL entry for X allows the type of access he's requesting. 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 disk. Unless it's cached, we'll need to read it off the disk 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 disk 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 disk do we put the ACL to ensure that we at least don't also have to do another seek? It had better be close to 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 naïve 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 of his 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 naïve 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 9 bits for each file's ACL. 9 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 9 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 away in the inode. One is the members of a particular group or users; this group ID could also be stored in the inode. The final one is everybody else, 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 bit of 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 9 bit ACLs<sup>3</sup>.

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

---

<sup>3</sup> 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 crossbreeding of these even earlier systems.

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.

#### ASIDE: NAME SPACES

We just encountered one of the interesting and difficult problems in distributed systems: what do names mean on different machines? The 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.

## 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 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 [DV64] being perhaps the earliest example. [W+74] describes the Hydra Operating System, which used capabilities as a fundamental control mechanism. [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 off-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 our 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 in various places. 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 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 on whether its security policy permits B to access `/tmp/foo`, and to 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 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 some high overheads. If we used capabilities for file-based access control, a user might have tens or hundreds of thousands of capabilities, one for each file he 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 a user has at his discretion to give to each process he runs.

There is another option. Capabilities need not be stored in the operating system. Instead, they can be cryptographically protected entities, which solves the forgeability problem and allows them to be left in users' 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 his capability list. Revoking his access merely requires removing the capability from the list, which is easy enough if the operating system 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' 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 only 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 has ACL-based access control performed. 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' 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.

## 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 himself 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 access control elements, which allow further restriction (but not loosening) of the 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. [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.

## 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.

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 and services. Or, 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 organizations. 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 the company determines that all programmers should be granted access to a new library that has been developed, but accountants and janitors 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 (or, less happily, demoted to janitor), you can merely remove the *Programmer* role from the set he could take on.

RBAC sounds a bit like using groups in access control lists, and there is some similarity, but RBAC systems typically have a more formal approach than merely including individuals in groups. They 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. RBAC systems may offer finer granularity than merely being able to read or write a file. A particular role (*Salesman*, 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 restocking record for the same product to the same file, since salesmen don't do restocking. 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. There are implications for performance, storage of the security context information, and authentication that we hope are obvious to you, at this stage.

One can build a minimal RBAC system under Linux and similar OSes using ACLs and groups. The Linux `sudo` command offers a simple approach, allowing users with particular privileges to run commands under other identities. 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 the user who ran the command was on a system-maintained list of users allowed to take on the identity *Programmer*. Usually the `sudo` command requires a new authentication step, as with other RBAC systems.

For more advanced purposes, one typically uses a system that supports finer granularity and more careful tracking of role assignment. This 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, he could take on the role of Programmer to do the installation himself, but could not use that role to allow salesmen or accountants to perform the installation.

#### 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 of them are malicious. Further, most apps have no legitimate need for most of the resources stored 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 thus implies that we should not give apps the full privileges belonging to the phone's owner. But they must have some privileges if they are to do anything interesting for that user.

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 his device, he is shown what permissions it requires. He can either grant the app those permissions, not install the app, or limit its permissions, though the latter choice may also limit the app's utility. Also, the developer specifies ways in which other apps can communicate with his 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 the app's 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 that possession 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, though, again, limiting them may damage app functionality. Permission labels are a form of mandatory access control. The Android security model is discussed in detail in [E+09].

The Android security approach is interesting, but it is 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. Which is too bad if the app is malicious.

## 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”  
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”  
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”  
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 here.*

[FK92] “Role-Based Access Controls”

David Ferraiolo and D. Richard Kuhn

15<sup>th</sup> 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*

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"

Peter Lescocco

Proceedings of the FREENIX Track:... 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 do so.*

[S74] “Protection and Control of Information Sharing in Multics”

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”

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 . . .*

[W+74] “Hydra: The Kernel of a Multiprocessor Operating System”

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.*

# Protecting Information With Cryptography

## 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 it is given control of. 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 he obtains it, our secrets are safe. Further, if he cannot understand it, he probably can't alter it, at least not in a controllable way. If he doesn't know what it means in its current form, how can he know how to change it into something he'd prefer?

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 the opponent from altering the data to suit his desires, and know when an opponent has tampered with our data. All through the joys of cryptography!

## THE CRUX OF THE PROBLEM

### HOW TO PROTECT INFORMATION OUTSIDE THE OPERATING SYSTEM'S DOMAIN?

How can we ensure that, even if others gain access to critical data outside the control of the operating system, they will neither be able to use nor alter it? What 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?

## 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<sup>1</sup>. 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.

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, 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 second piece of information (which is called a *key*)  $K$ , and an encryption algorithm  $E()$ . We end up with  $C$ , the altered form of  $P$ , which we usually call the *ciphertext*:

$$C = E(P, K)$$

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

---

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

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  $K$ :

$$P=D(C,K)$$

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 he still won’t be able to learn what  $P$  is.

Provided, of course, . . .

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 he does, he’ll 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, he can't understand it. But what if he can alter it? What he'll be altering is the encrypted form, so he'll make 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 his 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 in 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.

#### ASIDE: DEVELOPING YOUR OWN CIPHERS

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. 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.

So, don't.

To be particularly careful, we can use a *cryptographic hash* for this purpose. Cryptographic hashes are designed to make it computationally infeasible to come up with two plaintexts that hash to the same value, so use of such a hash gives us extra assurance that our data hasn't been tampered with. In fact, if we only care about integrity, rather than secrecy, we can take the cryptographic hash of a piece of data, encrypt only the hash, and send both the encrypted hash and the 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<sup>2</sup>.

Cryptographic hashes can be used for other purposes, as well. 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.

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

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.

This form of cryptography 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.

## 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

---

<sup>2</sup> Why do we need to encrypt the cryptographic hash? Well, anyone, including our opponent, can run a cryptographic hashing algorithm on anything, including his altered version of the message. If we don't encrypt the hash, he'll simply change the message, compute a new hash, replace both the original message and the original hash with his versions, and send the result. If the hash we sent is encrypted, though, he can't know what the encrypted version of the altered hash should be.

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

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

And our decryption operation becomes

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

Life has just become a lot easier for Microsoft. They can tell everyone their decryption key  $K_{\text{decrypt}}$ , but keep their encryption key  $K_{\text{encrypt}}$  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  $K_{\text{encrypt}}$  and send it out to all their users. Each user could decrypt it with  $K_{\text{decrypt}}$ . 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  $K_{\text{decrypt}}$  to encrypt, in which case you need the encryption key  $K_{\text{encrypt}}$  to decrypt. We still expect the encryption key to be kept secret and the decryption key to be publically known, so doing things in this order no longer allows authentication. Anyone could encrypt with  $K_{\text{decrypt}}$ , after all. But only the owner of the key can decrypt such messages using  $K_{\text{encrypt}}$ . 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 particular things it's so 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, then messages created by Eve would be misidentified by Bob as originating from Alice, totally subverting our entire goal of authenticating the messages. So we'd better make sure Eve can't fool Bob about which public key belongs to Alice.

This leads down a long and rather 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 talk a bit more about how, in practice, we distribute public keys in our 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?

## 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?

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]. 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. This flaw allowed attackers to read the traffic of something between  $\frac{1}{4}$  and  $\frac{1}{2}$  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 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.

As another example, an early implementation of the Netscape web browser generated cryptographic keys using some easily guessable 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 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 know 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.

#### 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.

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 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`. Use it to generate your keys.

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. So . . .

THE CRYPTOGRAPHY'S BENEFIT RELIES ENTIRELY ON THE SECRECY OF THE KEY.

## Cryptography and Operating Systems

Cryptography is fascinating, but lots of things are fascinating, 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.

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 the data to them in plaintext and asking them to encrypt it. They've seen the plaintext and can keep a copy of the key<sup>3</sup>.

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

---

<sup>3</sup> 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.

cryptography will not protect you, since that ever-so-vital secrecy of the key will be compromised.

So if cryptography won't protect us against a dishonest operating system, what's operating system 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 operating system is compromised, since the encrypted passwords can't be decrypted. (But if the legitimate user ever provides the correct password to a compromised operating system, all bets are off, alas. The compromised operating system 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.)

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 final destination machine will need the key if 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.

## 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. 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 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. So, for example, it offers little protection in the face of buffer overflow or SQL injection attacks.
- It does not protect against dishonest privileged users on the system, such as a system administrator. If his privileges allow him to pose as the user who owns the data or to install system components that give him access to the user's data, he will be given decrypted copies of the data on request.
- It does not protect against security flaws in the operating system 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 operating system 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.
- 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), 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). 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 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 him 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 he needed a password. 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 his password when he first accesses the system, but is not required to re-authenticate himself again until he logs 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.

## 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. Strong ciphers make it computationally infeasible to obtain the original bit pattern without access to the required key.

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.

## References

- [D88] “The First Ten Years of Public Key Cryptography”  
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”  
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”  
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.*
- [GW96] “Randomness and the Netscape Browser”  
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?*
- [K96 ] *The Codebreakers*  
David Kahn  
Scribner Publishing, 1996.  
*A long history of cryptography, its uses, and how it is attacked.*
- [S96] *Applied Cryptography*  
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”  
Stefan Viehböck  
<http://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.*

# Distributed Systems Goals & Challenges

Before we start discussing distributed systems architectures it is important to understand why we have been driven to build distributed systems, and the fundamental problems associated with doing so.

## Goals: Why Build Distributed Systems

Distributed systems could easily be justified by the simple facts of collaboration and sharing. The world-wide web is an obvious and compelling example of the value that is created when people can easily expose and exchange information. Much of the work we do is done in collaboration with others, and so we need often need to share work products. But there are several other powerful forces driving us towards distributed systems.

### Client/Server Usage Model

Long ago, all the equipment I needed to use in connection with my computer was connected to my computer. But this does not always make sense:

- I may only use a high resolution color scanner for a few minutes per month.
- I may make regular use of a high speed color printer, but not enough to justify buying one just for myself.
- I could store my music and videos on my own computer, but if I store them on a home NAS server, the entire family can have access to our combined libraries.
- I could store all my work related files on my own computer, but if I store them on a work-group server, somebody else will manage the back-ups, and ensure that everyone on the project has access to them.

There are many situations where we can get better functionality and save money by using remote/centralized resources rather than requiring all resources to be connected to a client computer.

### Reliability and Availability

As we come to depend more and more on our digital computers and storage we require higher reliability and availability from them. Long ago people tried to improve reliability by building systems out of the best possible components. But, as with RAID, we have learned that we can obtain better reliability by combining multiple (very ordinary systems). If we have more computing and storage capacity than we actually need, we may be able to continue providing service, even after one or more of our servers have failed.

The key is to distribute service over multiple independent servers. The reason they must be independent is so that they have no *single point of failure* ... no single component whose failure would take out multiple systems. If the client and server instances are to be distributed across multiple independent computers, then we are building a distributed system.

### Scalability

It is common to start any new project on a small system. If the system is successful, we will probably add more work to it over time. This means we will need more storage capacity, more network bandwidth, and more computing power. System manufacturers would be delighted if, each time we needed more capacity and power, we bought a new (larger, more expensive) computer (and threw away the old one). But

- a. This is highly inefficient, as we are essentially throwing away old capacity in order to buy new capacity.
- b. If we are successful, our needs for capacity and power will eventually exceed even the largest computer.

A more practical approach would be to design systems that can be expanded incrementally, by adding additional computers and storage as they were needed. And, again, if our growth plan is to *scale-out* (rather than *scale-up*) we are going to be building our system out of multiple independent computers, and so we are building a distributed system.

## Flexibility

We may start building and testing all the parts of a new service on a notebook or desktop, but later we may decide that we need to run different parts on different computers, or a single part on multiple computers. If our the components of our service interact with one-another through network protocols, it will likely be very easy to change the deployment model (which services run on which computers). Distributed systems tend to be very flexible in this respect.

## Challenges: Why are Distributed Systems Hard to Build

The short answer is that there are two reasons:

- Many solutions that work on single systems, do not work in distributed systems.
- Distributed systems have new problems that were never encountered in single systems.

## New and More Modes of Failure

If something bad happens to a single system (e.g. the failure of a disk or power supply) the whole system goes down. Having all the software fail at the same time is bad for service availability, but we don't have to worry about how some components can continue operating after others have failed. Partial failures are common in distributed systems:

- one node can crash while others continue running
- occasional network messages may be delayed or lost
- a switch failure may interrupt communication between some nodes, but not others

Distributed systems introduce many new problems that we might never have been forced to address in single systems:

- In a single system it may be very easy to tell that one of the service processes has died (e.g. the process exited with a fatal signal or error return code). In a distributed system our only indication that a component has failed might be that we are no longer receiving messages from it. Perhaps it has failed, or perhaps it is only slow, or perhaps the network link has failed, or perhaps our own network interface has failed. Problems are much more difficult to diagnose in a distributed system, and if we incorrectly diagnose a problem we are likely to choose the wrong solution.
- If we expect a distributed system to continue operating despite the failures of individual components, all of the components need to be made more robust (eg. greater error checking, automatic fail-over, recovery and connection reestablishment). One particularly tricky part of recovery is how to handle situations where a failed component was holding resource locks. We must find some way of recognizing the problem and breaking the locks. And after we have broken the locks we need some way of (a) restoring the resource to a clean state and (b) preventing the previous owner from attempting to continue using the resource if he returns.

## Complexity of Distributed State

Within a single computer system all system resource updates are correctly serialized and we can:

- place all operations on a single time-line (a total ordering)

- at any moment, say what the state of every resource in the system is.

Neither of these is true in a distributed system:

- Distinct nodes in a distributed system operate completely independently of one-another. Unless operations are performed by message exchanges, it is generally not possible to say whether a particular operation on node A happened before or after a different operation on node B. And even when operations are performed via message exchanges, two nodes may disagree on the relative ordering of two events (depending on the order in which each node received the messages).
- Because of the independence of parallel events, different nodes may at any given instant, consider a single resource to be in different states. Thus a resource does not actually have a single state. Rather its state is a vector of the state that the resource is considered to be in by each node in the system.

In single systems, when we needed before-or-after atomicity, we created a single mutex (perhaps in the operating system, or in memory shared by all contending threads). A similar effect can be achieved by sending messages to a central coordinator ... except that those messages are roughly a million times as expensive as operations on an in-memory mutex. This means that serialization approaches that worked very well in a single system can become prohibitively expensive in a distributed system.

## **Complexity of Management**

In a single computer system has a single configuration. A thousand different systems may each be configured differently:

- they may have different databases of known users
- their services may be configured with different options
- they may have different lists of which servers perform which functions
- their switches may be configured with different routing and fire-wall rules

And even if we create a distributed management service to push management updates out to all nodes:

- some nodes may not be up when the updates are sent, and so not learn of them
- networking problems may create isolated islands of nodes that are operating with a different configuration

## **Much Higher Loads**

One of the reasons we build distributed systems is to handle increasing loads. Higher loads often uncover weaknesses that had never caused problems under lighter loads. When a load increases by more than a power of ten, it is common to discover new bottlenecks. More nodes mean more messages, which may result in increased overhead, and longer delays. Increased overhead may result in poor scaling, or even in performance that drops as the system size grows. Longer (and more variable) delays often turn up race-conditions that had previously been highly unlikely.

## **Heterogeneity**

In a single computer system, all of the applications:

- are running on the same instruction set architecture
- are running on the same version of the same operating system
- are using the same versions of the same libraries
- directly interact with one-another through the operating system

In a distributed system, each node may be:

- a different instruction set architecture
- running a different operating system
- running different versions of the software and protocols

and the components interact with one-another through a variety of different networks and file systems. The combinatorics and constant evolution of possible component versions and interconnects render exhaustive testing impossible. These challenges often give rise to interoperability problems and unfortunate interactions that would never happen in a single (homogeneous) system.

## **Emergent phenomena**

The human mind renders complex systems understandable by constructing simpler abstract models. But simple models (almost by definition) cannot fully capture the behavior of a complex system. Complex systems often exhibit *emergent behaviors* that were not present in the constituent components, but arise from their interactions at scale (e.g. delay-induced oscillations in under-damped feed-back loops). If these phenomena do not happen in smaller systems, we can only learn about them through (hard) experience.

## Distributed Systems

Distributed systems have changed the face of the world. When your web browser connects to a web server somewhere else on the planet, it is participating in what seems to be a simple form of a **client/server** distributed system. When you contact a modern web service such as Google or Facebook, you are not just interacting with a single machine, however; behind the scenes, these complex services are built from a large collection (i.e., thousands) of machines, each of which cooperate to provide the particular service of the site. Thus, it should be clear what makes studying distributed systems interesting. Indeed, it is worthy of an entire class; here, we just introduce a few of the major topics.

A number of new challenges arise when building a distributed system. The major one we focus on is **failure**; machines, disks, networks, and software all fail from time to time, as we do not (and likely, will never) know how to build “perfect” components and systems. However, when we build a modern web service, we’d like it to appear to clients as if it never fails; how can we accomplish this task?

### THE CRUX:

#### HOW TO BUILD SYSTEMS THAT WORK WHEN COMPONENTS FAIL

How can we build a working system out of parts that don’t work correctly all the time? The basic question should remind you of some of the topics we discussed in RAID storage arrays; however, the problems here tend to be more complex, as are the solutions.

Interestingly, while failure is a central challenge in constructing distributed systems, it also represents an opportunity. Yes, machines fail; but the mere fact that a machine fails does not imply the entire system must fail. By collecting together a set of machines, we can build a system that appears to rarely fail, despite the fact that its components fail regularly. This reality is the central beauty and value of distributed systems, and why they underly virtually every modern web service you use, including Google, Facebook, etc.

**TIP: COMMUNICATION IS INHERENTLY UNRELIABLE**

In virtually all circumstances, it is good to view communication as a fundamentally unreliable activity. Bit corruption, down or non-working links and machines, and lack of buffer space for incoming packets all lead to the same result: packets sometimes do not reach their destination. To build reliable services atop such unreliable networks, we must consider techniques that can cope with packet loss.

Other important issues exist as well. System **performance** is often critical; with a network connecting our distributed system together, system designers must often think carefully about how to accomplish their given tasks, trying to reduce the number of messages sent and further make communication as efficient (low latency, high bandwidth) as possible.

Finally, **security** is also a necessary consideration. When connecting to a remote site, having some assurance that the remote party is who they say they are becomes a central problem. Further, ensuring that third parties cannot monitor or alter an on-going communication between two others is also a challenge.

In this introduction, we'll cover the most basic new aspect that is new in a distributed system: **communication**. Namely, how should machines within a distributed system communicate with one another? We'll start with the most basic primitives available, messages, and build a few higher-level primitives on top of them. As we said above, failure will be a central focus: how should communication layers handle failures?

## 47.1 Communication Basics

The central tenet of modern networking is that communication is fundamentally unreliable. Whether in the wide-area Internet, or a local-area high-speed network such as Infiniband, packets are regularly lost, corrupted, or otherwise do not reach their destination.

There are a multitude of causes for packet loss or corruption. Sometimes, during transmission, some bits get flipped due to electrical or other similar problems. Sometimes, an element in the system, such as a network link or packet router or even the remote host, are somehow damaged or otherwise not working correctly; network cables do accidentally get severed, at least sometimes.

More fundamental however is packet loss due to lack of buffering within a network switch, router, or endpoint. Specifically, even if we could guarantee that all links worked correctly, and that all the components in the system (switches, routers, end hosts) were up and running as expected, loss is still possible, for the following reason. Imagine a packet arrives at a router; for the packet to be processed, it must be placed in memory somewhere within the router. If many such packets arrive at

```
// client code
int main(int argc, char *argv[]) {
    int sd = UDP_Open(20000);
    struct sockaddr_in addrSnd, addrRcv;
    int rc = UDP_FillSockAddr(&addrSnd, "machine.cs.wisc.edu", 10000);
    char message[BUFFER_SIZE];
    sprintf(message, "hello world");
    rc = UDP_Write(sd, &addrSnd, message, BUFFER_SIZE);
    if (rc > 0) {
        int rc = UDP_Read(sd, &addrRcv, message, BUFFER_SIZE);
    }
    return 0;
}

// server code
int main(int argc, char *argv[]) {
    int sd = UDP_Open(10000);
    assert(sd > -1);
    while (1) {
        struct sockaddr_in addr;
        char message[BUFFER_SIZE];
        int rc = UDP_Read(sd, &addr, message, BUFFER_SIZE);
        if (rc > 0) {
            char reply[BUFFER_SIZE];
            sprintf(reply, "goodbye world");
            rc = UDP_Write(sd, &addr, reply, BUFFER_SIZE);
        }
    }
    return 0;
}
```

Figure 47.1: Example UDP/IP Client/Server Code

once, it is possible that the memory within the router cannot accommodate all of the packets. The only choice the router has at that point is to **drop** one or more of the packets. This same behavior occurs at end hosts as well; when you send a large number of messages to a single machine, the machine's resources can easily become overwhelmed, and thus packet loss again arises.

Thus, packet loss is fundamental in networking. The question thus becomes: how should we deal with it?

## 47.2 Unreliable Communication Layers

One simple way is this: we don't deal with it. Because some applications know how to deal with packet loss, it is sometimes useful to let them communicate with a basic unreliable messaging layer, an example of the **end-to-end argument** one often hears about (see the **Aside** at end of chapter). One excellent example of such an unreliable layer is found in the **UDP/IP** networking stack available today on virtually all modern systems. To use UDP, a process uses the **sockets** API in order to create a **communication endpoint**; processes on other machines (or on the same machine) send UDP **datagrams** to the original process (a datagram is a fixed-sized message up to some max size).

```

int UDP_Open(int port) {
    int sd;
    if ((sd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) { return -1; }
    struct sockaddr_in myaddr;
    bzero(&myaddr, sizeof(myaddr));
    myaddr.sin_family      = AF_INET;
    myaddr.sin_port        = htons(port);
    myaddr.sin_addr.s_addr = INADDR_ANY;
    if (bind(sd, (struct sockaddr *) &myaddr, sizeof(myaddr)) == -1) {
        close(sd);
        return -1;
    }
    return sd;
}

int UDP_FillSockAddr(struct sockaddr_in *addr, char *hostName, int port) {
    bzero(addr, sizeof(struct sockaddr_in));
    addr->sin_family = AF_INET;           // host byte order
    addr->sin_port   = htons(port);       // short, network byte order
    struct in_addr *inAddr;
    struct hostent *hostEntry;
    if ((hostEntry = gethostbyname(hostName)) == NULL) { return -1; }
    inAddr = (struct in_addr *) hostEntry->h_addr;
    addr->sin_addr = *inAddr;
    return 0;
}

int UDP_Write(int sd, struct sockaddr_in *addr, char *buffer, int n) {
    int addrLen = sizeof(struct sockaddr_in);
    return sendto(sd, buffer, n, 0, (struct sockaddr *) addr, addrLen);
}

int UDP_Read(int sd, struct sockaddr_in *addr, char *buffer, int n) {
    int len = sizeof(struct sockaddr_in);
    return recvfrom(sd, buffer, n, 0, (struct sockaddr *) addr,
                   (socklen_t *) &len);
}

```

Figure 47.2: A Simple UDP Library

Figures 47.1 and 47.2 show a simple client and server built on top of UDP/IP. The client can send a message to the server, which then responds with a reply. With this small amount of code, you have all you need to begin building distributed systems!

UDP is a great example of an unreliable communication layer. If you use it, you will encounter situations where packets get lost (dropped) and thus do not reach their destination; the sender is never thus informed of the loss. However, that does not mean that UDP does not guard against any failures at all. For example, UDP includes a **checksum** to detect some forms of packet corruption.

However, because many applications simply want to send data to a destination and not worry about packet loss, we need more. Specifically, we need reliable communication on top of an unreliable network.

**TIP: USE CHECKSUMS FOR INTEGRITY**

Checksums are a commonly-used method to detect corruption quickly and effectively in modern systems. A simple checksum is addition: just sum up the bytes of a chunk of data; of course, many other more sophisticated checksums have been created, including basic cyclic redundancy codes (CRCs), the Fletcher checksum, and many others [MK09].

In networking, checksums are used as follows. Before sending a message from one machine to another, compute a checksum over the bytes of the message. Then send both the message and the checksum to the destination. At the destination, the receiver computes a checksum over the incoming message as well; if this computed checksum matches the sent checksum, the receiver can feel some assurance that the data likely did not get corrupted during transmission.

Checksums can be evaluated along a number of different axes. Effectiveness is one primary consideration: does a change in the data lead to a change in the checksum? The stronger the checksum, the harder it is for changes in the data to go unnoticed. Performance is the other important criterion: how costly is the checksum to compute? Unfortunately, effectiveness and performance are often at odds, meaning that checksums of high quality are often expensive to compute. Life, again, isn't perfect.

### 47.3 Reliable Communication Layers

To build a reliable communication layer, we need some new mechanisms and techniques to handle packet loss. Let us consider a simple example in which a client is sending a message to a server over an unreliable connection. The first question we must answer: how does the sender know that the receiver has actually received the message?

The technique that we will use is known as an **acknowledgment**, or **ack** for short. The idea is simple: the sender sends a message to the receiver; the receiver then sends a short message back to *acknowledge* its receipt. Figure 47.3 depicts the process.

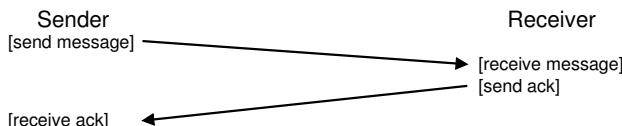


Figure 47.3: Message Plus Acknowledgment

When the sender receives an acknowledgment of the message, it can then rest assured that the receiver did indeed receive the original message. However, what should the sender do if it does not receive an acknowledgment?

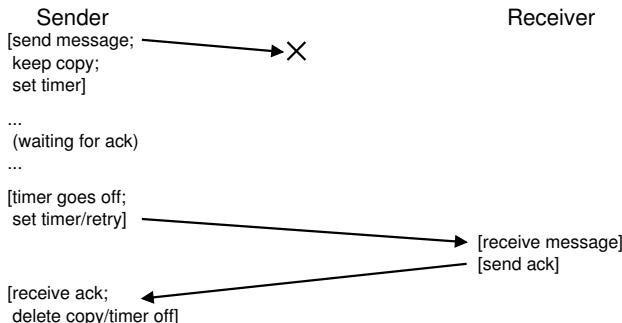


Figure 47.4: Message Plus Acknowledgment: Dropped Request

To handle this case, we need an additional mechanism, known as a **timeout**. When the sender sends a message, the sender now sets a timer to go off after some period of time. If, in that time, no acknowledgment has been received, the sender concludes that the message has been lost. The sender then simply performs a **retry** of the send, sending the same message again with hopes that this time, it will get through. For this approach to work, the sender must keep a copy of the message around, in case it needs to send it again. The combination of the timeout and the retry have led some to call the approach **timeout/retry**; pretty clever crowd, those networking types, no? Figure 47.4 shows an example.

Unfortunately, timeout/retry in this form is not quite enough. Figure 47.5 shows an example of packet loss which could lead to trouble. In this example, it is not the original message that gets lost, but the acknowledgment. From the perspective of the sender, the situation seems the same: no ack was received, and thus a timeout and retry are in order. But from the perspective of the receiver, it is quite different: now the same message has been received twice! While there may be cases where this is OK, in general it is not; imagine what would happen when you are downloading a file and extra packets are repeated inside the download. Thus, when we are aiming for a reliable message layer, we also usually want to guarantee that each message is received **exactly once** by the receiver.

To enable the receiver to detect duplicate message transmission, the sender has to identify each message in some unique way, and the receiver needs some way to track whether it has already seen each message before. When the receiver sees a duplicate transmission, it simply acks the message, but (critically) does *not* pass the message to the application that receives the data. Thus, the sender receives the ack but the message is not received twice, preserving the exactly-once semantics mentioned above.

There are myriad ways to detect duplicate messages. For example, the sender could generate a unique ID for each message; the receiver could track every ID it has ever seen. This approach could work, but it is prohibitively costly, requiring unbounded memory to track all IDs.

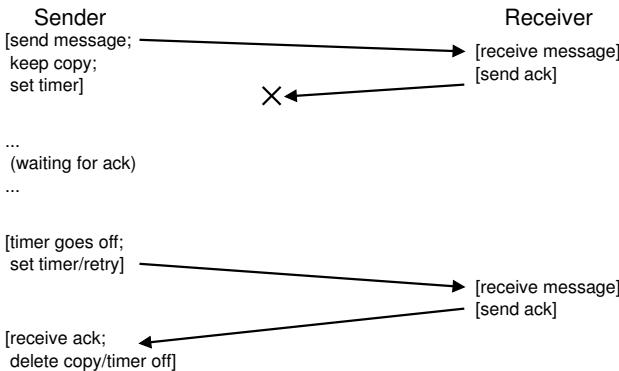


Figure 47.5: Message Plus Acknowledgment: Dropped Reply

A simpler approach, requiring little memory, solves this problem, and the mechanism is known as a **sequence counter**. With a sequence counter, the sender and receiver agree upon a start value (e.g., 1) for a counter that each side will maintain. Whenever a message is sent, the current value of the counter is sent along with the message; this counter value ( $N$ ) serves as an ID for the message. After the message is sent, the sender then increments the value (to  $N + 1$ ).

The receiver uses its counter value as the expected value for the ID of the incoming message from that sender. If the ID of a received message ( $N$ ) matches the receiver's counter (also  $N$ ), it acks the message and passes it up to the application; in this case, the receiver concludes this is the first time this message has been received. The receiver then increments its counter (to  $N + 1$ ), and waits for the next message.

If the ack is lost, the sender will timeout and re-send message  $N$ . This time, the receiver's counter is higher ( $N + 1$ ), and thus the receiver knows it has already received this message. Thus it acks the message but does *not* pass it up to the application. In this simple manner, sequence counters can be used to avoid duplicates.

The most commonly used reliable communication layer is known as **TCP/IP**, or just **TCP** for short. TCP has a great deal more sophistication than we describe above, including machinery to handle congestion in the network [VJ88], multiple outstanding requests, and hundreds of other small tweaks and optimizations. Read more about it if you're curious; better yet, take a networking course and learn that material well.

## 47.4 Communication Abstractions

Given a basic messaging layer, we now approach the next question in this chapter: what abstraction of communication should we use when building a distributed system?

**TIP: BE CAREFUL SETTING THE TIMEOUT VALUE**

As you can probably guess from the discussion, setting the timeout value correctly is an important aspect of using timeouts to retry message sends. If the timeout is too small, the sender will re-send messages needlessly, thus wasting CPU time on the sender and network resources. If the timeout is too large, the sender waits too long to re-send and thus perceived performance at the sender is reduced. The “right” value, from the perspective of a single client and server, is thus to wait just long enough to detect packet loss but no longer.

However, there are often more than just a single client and server in a distributed system, as we will see in future chapters. In a scenario with many clients sending to a single server, packet loss at the server may be an indicator that the server is overloaded. If true, clients might retry in a different adaptive manner; for example, after the first timeout, a client might increase its timeout value to a higher amount, perhaps twice as high as the original value. Such an **exponential back-off** scheme, pioneered in the early Aloha network and adopted in early Ethernet [A70], avoids situations where resources are being overloaded by an excess of re-sends. Robust systems strive to avoid overload of this nature.

The systems community developed a number of approaches over the years. One body of work took OS abstractions and extended them to operate in a distributed environment. For example, **distributed shared memory (DSM)** systems enable processes on different machines to share a large, virtual address space [LH89]. This abstraction turns a distributed computation into something that looks like a multi-threaded application; the only difference is that these threads run on different machines instead of different processors within the same machine.

The way most DSM systems work is through the virtual memory system of the OS. When a page is accessed on one machine, two things can happen. In the first (best) case, the page is already local on the machine, and thus the data is fetched quickly. In the second case, the page is currently on some other machine. A page fault occurs, and the page fault handler sends a message to some other machine to fetch the page, install it in the page table of the requesting process, and continue execution.

This approach is not widely in use today for a number of reasons. The largest problem for DSM is how it handles failure. Imagine, for example, if a machine fails; what happens to the pages on that machine? What if the data structures of the distributed computation are spread across the entire address space? In this case, parts of these data structures would suddenly become unavailable. Dealing with failure when parts of your address space go missing is hard; imagine a linked list that where a next pointer points into a portion of the address space that is gone. Yikes!

A further problem is performance. One usually assumes, when writing code, that access to memory is cheap. In DSM systems, some accesses

are inexpensive, but others cause page faults and expensive fetches from remote machines. Thus, programmers of such DSM systems had to be very careful to organize computations such that almost no communication occurred at all, defeating much of the point of such an approach. Though much research was performed in this space, there was little practical impact; nobody builds reliable distributed systems using DSM today.

## 47.5 Remote Procedure Call (RPC)

While OS abstractions turned out to be a poor choice for building distributed systems, programming language (PL) abstractions make much more sense. The most dominant abstraction is based on the idea of a **remote procedure call**, or **RPC** for short [BN84]<sup>1</sup>.

Remote procedure call packages all have a simple goal: to make the process of executing code on a remote machine as simple and straightforward as calling a local function. Thus, to a client, a procedure call is made, and some time later, the results are returned. The server simply defines some routines that it wishes to export. The rest of the magic is handled by the RPC system, which in general has two pieces: a **stub generator** (sometimes called a **protocol compiler**), and the **run-time library**. We'll now take a look at each of these pieces in more detail.

### Stub Generator

The stub generator's job is simple: to remove some of the pain of packing function arguments and results into messages by automating it. Numerous benefits arise: one avoids, by design, the simple mistakes that occur in writing such code by hand; further, a stub compiler can perhaps optimize such code and thus improve performance.

The input to such a compiler is simply the set of calls a server wishes to export to clients. Conceptually, it could be something as simple as this:

```
interface {
    int func1(int arg1);
    int func2(int arg1, int arg2);
};
```

The stub generator takes an interface like this and generates a few different pieces of code. For the client, a **client stub** is generated, which contains each of the functions specified in the interface; a client program wishing to use this RPC service would link with this client stub and call into it in order to make RPCs.

Internally, each of these functions in the client stub do all of the work needed to perform the remote procedure call. To the client, the code just

---

<sup>1</sup>In modern programming languages, we might instead say **remote method invocation (RMI)**, but who likes these languages anyhow, with all of their fancy objects?

appears as a function call (e.g., the client calls `func1(x)`); internally, the code in the client stub for `func1()` does this:

- **Create a message buffer.** A message buffer is usually just a contiguous array of bytes of some size.
- **Pack the needed information into the message buffer.** This information includes some kind of identifier for the function to be called, as well as all of the arguments that the function needs (e.g., in our example above, one integer for `func1`). The process of putting all of this information into a single contiguous buffer is sometimes referred to as the **marshaling** of arguments or the **serialization** of the message.
- **Send the message to the destination RPC server.** The communication with the RPC server, and all of the details required to make it operate correctly, are handled by the RPC run-time library, described further below.
- **Wait for the reply.** Because function calls are usually **synchronous**, the call will wait for its completion.
- **Unpack return code and other arguments.** If the function just returns a single return code, this process is straightforward; however, more complex functions might return more complex results (e.g., a list), and thus the stub might need to unpack those as well. This step is also known as **unmarshaling** or **deserialization**.
- **Return to the caller.** Finally, just return from the client stub back into the client code.

For the server, code is also generated. The steps taken on the server are as follows:

- **Unpack the message.** This step, called **unmarshaling** or **deserialization**, takes the information out of the incoming message. The function identifier and arguments are extracted.
- **Call into the actual function.** Finally! We have reached the point where the remote function is actually executed. The RPC runtime calls into the function specified by the ID and passes in the desired arguments.
- **Package the results.** The return argument(s) are marshaled back into a single reply buffer.
- **Send the reply.** The reply is finally sent to the caller.

There are a few other important issues to consider in a stub compiler. The first is complex arguments, i.e., how does one package and send a complex data structure? For example, when one calls the `write()` system call, one passes in three arguments: an integer file descriptor, a pointer to a buffer, and a size indicating how many bytes (starting at the pointer) are to be written. If an RPC package is passed a pointer, it needs to be able to figure out how to interpret that pointer, and perform the

correct action. Usually this is accomplished through either well-known types (e.g., a `buffer_t` that is used to pass chunks of data given a size, which the RPC compiler understands), or by annotating the data structures with more information, enabling the compiler to know which bytes need to be serialized.

Another important issue is the organization of the server with regards to concurrency. A simple server just waits for requests in a simple loop, and handles each request one at a time. However, as you might have guessed, this can be grossly inefficient; if one RPC call blocks (e.g., on I/O), server resources are wasted. Thus, most servers are constructed in some sort of concurrent fashion. A common organization is a **thread pool**. In this organization, a finite set of threads are created when the server starts; when a message arrives, it is dispatched to one of these worker threads, which then does the work of the RPC call, eventually replying; during this time, a main thread keeps receiving other requests, and perhaps dispatching them to other workers. Such an organization enables concurrent execution within the server, thus increasing its utilization; the standard costs arise as well, mostly in programming complexity, as the RPC calls may now need to use locks and other synchronization primitives in order to ensure their correct operation.

## Run-Time Library

The run-time library handles much of the heavy lifting in an RPC system; most performance and reliability issues are handled herein. We'll now discuss some of the major challenges in building such a run-time layer.

One of the first challenges we must overcome is how to locate a remote service. This problem, of **naming**, is a common one in distributed systems, and in some sense goes beyond the scope of our current discussion. The simplest of approaches build on existing naming systems, e.g., hostnames and port numbers provided by current internet protocols. In such a system, the client must know the hostname or IP address of the machine running the desired RPC service, as well as the port number it is using (a port number is just a way of identifying a particular communication activity taking place on a machine, allowing multiple communication channels at once). The protocol suite must then provide a mechanism to route packets to a particular address from any other machine in the system. For a good discussion of naming, you'll have to look elsewhere, e.g., read about DNS and name resolution on the Internet, or better yet just read the excellent chapter in Saltzer and Kaashoek's book [SK09].

Once a client knows which server it should talk to for a particular remote service, the next question is which transport-level protocol should RPC be built upon. Specifically, should the RPC system use a reliable protocol such as TCP/IP, or be built upon an unreliable communication layer such as UDP/IP?

Naively the choice would seem easy: clearly we would like for a request to be reliably delivered to the remote server, and clearly we would

like to reliably receive a reply. Thus we should choose the reliable transport protocol such as TCP, right?

Unfortunately, building RPC on top of a reliable communication layer can lead to a major inefficiency in performance. Recall from the discussion above how reliable communication layers work: with acknowledgments plus timeout/retry. Thus, when the client sends an RPC request to the server, the server responds with an acknowledgment so that the caller knows the request was received. Similarly, when the server sends the reply to the client, the client acks it so that the server knows it was received. By building a request/response protocol (such as RPC) on top of a reliable communication layer, two “extra” messages are sent.

For this reason, many RPC packages are built on top of unreliable communication layers, such as UDP. Doing so enables a more efficient RPC layer, but does add the responsibility of providing reliability to the RPC system. The RPC layer achieves the desired level of responsibility by using timeout/retry and acknowledgments much like we described above. By using some form of sequence numbering, the communication layer can guarantee that each RPC takes place exactly once (in the case of no failure), or at most once (in the case where failure arises).

## Other Issues

There are some other issues an RPC run-time must handle as well. For example, what happens when a remote call takes a long time to complete? Given our timeout machinery, a long-running remote call might appear as a failure to a client, thus triggering a retry, and thus the need for some care here. One solution is to use an explicit acknowledgment (from the receiver to sender) when the reply isn’t immediately generated; this lets the client know the server received the request. Then, after some time has passed, the client can periodically ask whether the server is still working on the request; if the server keeps saying “yes”, the client should be happy and continue to wait (after all, sometimes a procedure call can take a long time to finish executing).

The run-time must also handle procedure calls with large arguments, larger than what can fit into a single packet. Some lower-level network protocols provide such sender-side **fragmentation** (of larger packets into a set of smaller ones) and receiver-side **reassembly** (of smaller parts into one larger logical whole); if not, the RPC run-time may have to implement such functionality itself. See Birrell and Nelson’s excellent RPC paper for details [BN84].

One issue that many systems handle is that of **byte ordering**. As you may know, some machines store values in what is known as **big endian** ordering, whereas others use **little endian** ordering. Big endian stores bytes (say, of an integer) from most significant to least significant bits, much like Arabic numerals; little endian does the opposite. Both are equally valid ways of storing numeric information; the question here is how to communicate between machines of *different* endianness.

### Aside: The End-to-End Argument

The **end-to-end argument** makes the case that the highest level in a system, i.e., usually the application at “the end”, is ultimately the only locale within a layered system where certain functionality can truly be implemented. In their landmark paper [SRC84], Saltzer et al. argue this through an excellent example: reliable file transfer between two machines. If you want to transfer a file from machine *A* to machine *B*, and make sure that the bytes that end up on *B* are exactly the same as those that began on *A*, you must have an “end-to-end” check of this; lower-level reliable machinery, e.g., in the network or disk, provides no such guarantee.

The contrast is an approach which tries to solve the reliable-file-transfer problem by adding reliability to lower layers of the system. For example, say we build a reliable communication protocol and use it to build our reliable file transfer. The communication protocol guarantees that every byte sent by a sender will be received in order by the receiver, say using timeout/retry, acknowledgments, and sequence numbers. Unfortunately, using such a protocol does not a reliable file transfer make; imagine the bytes getting corrupted in sender memory before the communication even takes place, or something bad happening when the receiver writes the data to disk. In those cases, even though the bytes were delivered reliably across the network, our file transfer was ultimately not reliable. To build a reliable file transfer, one must include end-to-end checks of reliability, e.g., after the entire transfer is complete, read back the file on the receiver disk, compute a checksum, and compare that checksum to that of the file on the sender.

The corollary to this maxim is that sometimes having lower layers provide extra functionality can indeed improve system performance or otherwise optimize a system. Thus, you should not rule out having such machinery at a lower-level in a system; rather, you should carefully consider the utility of such machinery, given its eventual usage in an overall system or application.

RPC packages often handle this by providing a well-defined endianness within their message formats. In Sun’s RPC package, the **XDR** (**eXternal Data Representation**) layer provides this functionality. If the machine sending or receiving a message matches the endianness of XDR, messages are just sent and received as expected. If, however, the machine communicating has a different endianness, each piece of information in the message must be converted. Thus, the difference in endianness can have a small performance cost.

A final issue is whether to expose the asynchronous nature of communication to clients, thus enabling some performance optimizations. Specifically, typical RPCs are made **synchronously**, i.e., when a client issues the procedure call, it must wait for the procedure call to return

before continuing. Because this wait can be long, and because the client may have other work it could be doing, some RPC packages enable you to invoke an RPC **asynchronously**. When an asynchronous RPC is issued, the RPC package sends the request and returns immediately; the client is then free to do other work, such as call other RPCs or other useful computation. The client at some point will want to see the results of the asynchronous RPC; it thus calls back into the RPC layer, telling it to wait for outstanding RPCs to complete, at which point return arguments can be accessed.

## 47.6 Summary

We have seen the introduction of a new topic, distributed systems, and its major issue: how to handle failure which is now a commonplace event. As they say inside of Google, when you have just your desktop machine, failure is rare; when you're in a data center with thousands of machines, failure is happening all the time. The key to any distributed system is how you deal with that failure.

We have also seen that communication forms the heart of any distributed system. A common abstraction of that communication is found in remote procedure call (RPC), which enables clients to make remote calls on servers; the RPC package handles all of the gory details, including timeout/retry and acknowledgment, in order to deliver a service that closely mirrors a local procedure call.

The best way to really understand an RPC package is of course to use one yourself. Sun's RPC system, using the stub compiler `rpcgen`, is a common one, and is widely available on systems today, including Linux. Try it out, and see what all the fuss is about.

## References

- [A70] "The ALOHA System — Another Alternative for Computer Communications"  
Norman Abramson  
The 1970 Fall Joint Computer Conference  
*The ALOHA network pioneered some basic concepts in networking, including exponential back-off and retransmit, which formed the basis for communication in shared-bus Ethernet networks for years.*
- [BN84] "Implementing Remote Procedure Calls"  
Andrew D. Birrell, Bruce Jay Nelson  
ACM TOCS, Volume 2:1, February 1984  
*The foundational RPC system upon which all others build. Yes, another pioneering effort from our friends at Xerox PARC.*
- [MK09] "The Effectiveness of Checksums for Embedded Control Networks"  
Theresa C. Maxino and Philip J. Koopman  
IEEE Transactions on Dependable and Secure Computing, 6:1, January '09  
*A nice overview of basic checksum machinery and some performance and robustness comparisons between them.*
- [LH89] "Memory Coherence in Shared Virtual Memory Systems"  
Kai Li and Paul Hudak  
ACM TOCS, 7:4, November 1989  
*The introduction of software-based shared memory via virtual memory. An intriguing idea for sure, but not a lasting or good one in the end.*
- [SK09] "Principles of Computer System Design"  
Jerome H. Saltzer and M. Frans Kaashoek  
Morgan-Kaufmann, 2009  
*An excellent book on systems, and a must for every bookshelf. One of the few terrific discussions on naming we've seen.*
- [SRC84] "End-To-End Arguments in System Design"  
Jerome H. Saltzer, David P. Reed, David D. Clark  
ACM TOCS, 2:4, November 1984  
*A beautiful discussion of layering, abstraction, and where functionality must ultimately reside in computer systems.*
- [VJ88] "Congestion Avoidance and Control"  
Van Jacobson  
SIGCOMM '88  
*A pioneering paper on how clients should adjust to perceived network congestion; definitely one of the key pieces of technology underlying the Internet, and a must read for anyone serious about systems, and for Van Jacobson's relatives because well relatives should read all of your papers.*

# Distributed Systems

Monday, November 21, 2016 6:12 PM

(Representational state transfer) RESTful interfaces

- [https://www.wikiwand.com/en/Representational\\_state\\_transfer](https://www.wikiwand.com/en/Representational_state_transfer)
- Representational state transfer (REST) or RESTful web services are one way of providing interoperability between computer systems on the Internet. REST-compliant web services allow requesting systems to access and manipulate textual representations of web resources using a uniform and predefined set of stateless operations. Other forms of web service exist, which expose their own arbitrary sets of operations such as WSDL and SOAP.[1] "Web resources" were first defined on the World Wide Web as documents or files identified by their URLs, but today they have a much more generic and abstract definition encompassing every thing or entity that can be identified, named, addressed or handled, in any way whatsoever, on the web. In a REST web service, requests made to a resource's URI will elicit a response that may be in XML, HTML, JSON or some other defined format. The response may confirm that some alteration has been made to the stored resource, and it may provide hypertext links to other related resources or collections of resources. Using HTTP, as is most common, the kind of operations available include those predefined by the HTTP verbs GET, POST, PUT, DELETE and so on. By making use of a stateless protocol and standard operations REST systems aim for fast performance, reliability, and the ability to grow, by using reused components that can be managed and updated without affecting the system as a whole, even while it is running
- Architectural constraints
  - Client-Server
    - Separation allows components to evolve independently supporting Internet-scale requirement
    - Improves portability of user interface across multiple platforms
    - Improves scalability by simplifying the server

## components

- Stateless
  - No client context stored on the server between requests
  - The session state is held in the client and can be transferred by the server to another service such as a database to maintain a persistent state for a period and allow authentication
  - Client begins sending requests when it is ready to make the transition to a new state
    - When requests are outstanding, client is considered to be in transition.
- Cacheable
  - Responses must be defined as cacheable or not to prevent clients from reusing stale or inappropriate data in response to further requests
  - Well-managed caching partially/completely eliminates some client-server interactions, improving scalability and performance
- Layered System
  - Intermediary servers may improve system scalability by enabling load balancing and by providing shared caches
    - May also enforce security policies
- Code on demand (optional)
  - Servers can temporarily extend or customize the functionality of a client by the transfer of executable code
    - Compiled components such as Java applets, javascript scripts
- Uniform Interface
  - Simplifies and decouples architecture, enabling each part to evolve independently
  - Constraints
    - Identification of resources
      - ◆ Resources are conceptually separate from the representations that are returned to the client
      - ◆ Server may send data from its database as HTML, XML, JSON, but may not be the server's internal representation
    - Manipulation of resources through representations
      - ◆ When client holds a representation of a

resource, including any metadata attached, it has enough information to modify or delete the resource.

- Self-descriptive messages
  - ◆ Each message includes enough information to describe how to process the message
- Hypermedia as the engine of application state
  - ◆ REST client should be able to use server-provided links dynamically to discover all available actions and resources it needs
  - ◆ As access proceeds the server responds with text that include hyperlinks to other actions that are currently available
  - ◆ No need for the client to be hard coded with information regarding the structure or dynamics of the REST service

## Lease

- [https://www.wikiwand.com/en/Lease\\_\(computer\\_science\)](https://www.wikiwand.com/en/Lease_(computer_science))
- In computer science, a **Lease** is a contract that gives its holder specified rights to some resource for a limited period. Because it is time-limited, a lease is an alternative to a lock for resource serialization.
- Motivation
  - A traditional resource lock is granted until it is explicitly released by the locking client process. Reasons why a lock might not be released include:
    - The client failed before releasing the resources
    - The client deadlocked while attempting to allocate another resource
    - The client was blocked or delayed for an unreasonable period
    - The client neglected to free the resource, perhaps due to a bug
    - The request to free the resource was lost
    - The resource manager failed or lost track of the resource stated
  - Any of these could end the availability of an important reusable resource until the system is reset. By contract, a lease is valid for a limited period, after which it automatically expires, making the

resource available for reallocation by a new client.

- Leases are commonly used in distributed systems for applications ranging from [DHCP address allocation](#) to [file locking](#), but they are not (by themselves) a complete solution:
- Problems
  - There must be some means of notifying the lease holder of the expiration and preventing that agent from continuing to rely on the resource. Often, this is done by requiring all requests to be accompanied by an [access token](#), which is invalidated if the associated lease has expired.
  - If a lease is revoked after the lease holder has started operating on the resource, revocation may leave the resource in a compromised state. In such situations, it is common to use [Atomic transactions](#) to ensure that updates that do not complete have no effect.

## Consensus

- [https://www.wikiwand.com/en/Consensus\\_\(computer\\_science\)](https://www.wikiwand.com/en/Consensus_(computer_science))
- A fundamental problem in [distributed computing](#) and [multi-agent systems](#) is to achieve overall system reliability in the presence of a number of faulty processes. This often requires processes to agree on some data value that is needed during computation.
  - Examples of applications of **consensus** include
    - whether to commit a transaction to a database,
    - agreeing on the identity of a [leader](#),
    - [state machine replication](#),
    - [atomic broadcasts](#).
- One approach to generating consensus is for all processes (agents) to agree on a majority value. In this context, a majority requires at least one more than half of available votes (where each process is given a vote).
  - However one or more faulty processes may skew the resultant outcome such that consensus may not be reached or reached incorrectly.
- Protocols that solve consensus problems are designed to deal with limited numbers of faulty processes
  - Protocols must satisfy a number of requirements messng with input and output values.
- Properties of protocol
  - Termination
    - Every correct process decides some value
  - Validity

- If all processes propose the same value  $v$  then all correct processes decide  $v$
- Integrity
  - Every correct process decides at most one value, and if it decides some value  $v$  then  $v$  must have been proposed by some process
- Agreement
  - Every correct process must agree on the same value

# Distributed System Security

## 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:

1. The other machines in the distributed system might not properly implement the security policies you want.
2. Machines in a distributed system communicate across a network that none of them fully control and that, generally, cannot be trusted.

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. So,

### THE CRUX OF THE PROBLEM 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?

## 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 pretty 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 the same factors 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<sup>1</sup>. In each case, you need to know something: either the password (or, better, a cryptographic hash of the password plus a salt) or the public key.

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 bit 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, since 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. So let's concentrate for

---

<sup>1</sup> 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.

the moment on how we handle getting that public key, either to use it directly or to 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 his attack was authenticated and encrypted.

## 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 with him, he can give us his public key in some form or another, in which case we can be pretty sure it's his. That's limiting, though, since we often interact with partners who we never see face to face. For that matter, whose "face" belongs to Amazon or Google?

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 his 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  $K_F$ . Frobazz Inc. pays big bucks to Acmesign Co., a widely trusted company whose business it is to sell certificates. 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) on Frobazz's name, public key  $K_F$ , and other information, producing hash  $H_F$ . Acmesign then encrypts  $H_F$  with its own private key,  $P_A$ , producing digital signature  $S_F$ . Finally, Acmesign combines all of the

information used to produce  $H_F$ , plus Acmesign’s own identity and the signature  $S_F$ , into the certificate  $C_F$ , which it hands over to Frobazz, presumably in exchange for a bunch of money. Remember,  $C_F$  is just a bunch of 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  $C_F$  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  $H_F'$ . Then the customer uses Acmesign’s public key to decrypt  $S_F$  (also in the certificate), obtaining  $H_F$ . If all is well,  $H_F$  equals  $H_F'$ , and now the customer knows that the public key in the certificate is indeed Frobazz’s. Public key authentication can proceed<sup>2</sup>. 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 he needs. From that point onward, he can simply store it and use it. If, for whatever reason, he loses it, he can either extract it again from the certificate (if that has been saved), or go through the process of obtaining the certificate all over 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 keep that party at arm’s length and 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.

---

<sup>2</sup> 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 him the certificate actually was Frobazz or knew Frobazz’s private key.

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 serious caveats here, 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 is that 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 replay 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.

## **Password Authentication for Distributed Systems**

The other common option used 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, so we don't need to keep promising you that we'll get to it.

## 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 damned 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). 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 and another using Triple DES, 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 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 symmetric 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-1 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 in each case 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, which include skipping authentication and hoping for the best (not generally a good option, but still supported as of TLS version 1.2). The most common method is for the client to obtain a certificate containing the server's public key and to use the public key in that certificate to verify the authenticity of the server's messages, typically by having the server send it to the client. 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 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.)

## 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're wrong. Two extremely smart cryptographers named Diffie and 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  $\text{mod } n$ ). 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  $X = g^x \text{ mod } n$ , and sends  $X$  to Bob. Bob independently chooses a large random integer, say  $y$ , calculates  $Y = g^y \text{ mod } n$ , 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.

Alice now computes  $k = Y^x \text{ mod } n$ , and Bob computes  $k = X^y \text{ mod } n$ . Alice and Bob get the same value  $k$  from these computations. Why? Well,  $Y^x \text{ mod } n = (g^y \text{ mod } n)^x \text{ mod } n$ , which in turn equals  $g^{yx} \text{ mod } n$ .  $X^y \text{ mod } n = (g^x \text{ mod } n)^y \text{ mod } n = g^{xy} \text{ mod } n$ , which is the same thing Alice got. So  $k$  is the same and is known to both Alice and Bob.

What about those eavesdroppers? They know  $g$ ,  $n$ ,  $X$ , and  $Y$ , but not  $x$  or  $y$ . If they compute  $k' = X^y \text{ mod } n$ , they get  $g^x \text{ mod } n^{g^y \text{ mod } n} \text{ mod } n$ , which is not equal to the  $k$  Alice and Bob calculated. They do have an approach to derive  $x$  or  $y$ , which would give them enough information to obtain  $k$ , but that approach requires them to computer a discrete logarithm. That's a solvable problem, but computationally infeasible for large numbers. So if the prime  $n$  is large (and meets other properties), the eavesdroppers are out of luck.

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.

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 publically 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. So 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.

## 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. 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 store 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 his password again.

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 he 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 her password except when absolutely necessary, or is it better to provide your user with improved security by frequently requiring proof of her identity? The point isn't that there is one correct answer 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 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 the 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 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 which 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].

## 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. These 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 in all these cases. 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, but that won't be much consolation if it succeeds.

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 remote interactions.

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, such as the use of authentication servers (such as Kerberos) and password-based authentication. 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.

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.

## 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 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 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

[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”

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.*

## Sun's Network File System (NFS)

One of the first uses of distributed client/server computing was in the realm of distributed file systems. In such an environment, there are a number of client machines and one server (or a few); the server stores the data on its disks, and clients request data through well-formed protocol messages. Figure 48.1 depicts the basic setup.

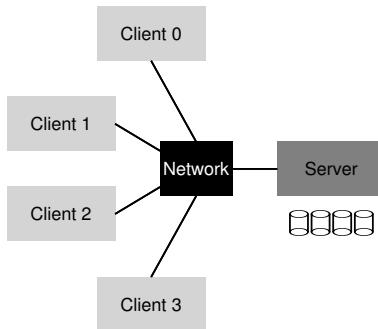


Figure 48.1: A Generic Client/Server System

As you can see from the picture, the server has the disks, and clients send messages across a network to access their directories and files on those disks. Why do we bother with this arrangement? (i.e., why don't we just let clients use their local disks?) Well, primarily this setup allows for easy **sharing** of data across clients. Thus, if you access a file on one machine (Client 0) and then later use another (Client 2), you will have the same view of the file system. Your data is naturally shared across these different machines. A secondary benefit is **centralized administration**; for example, backing up files can be done from the few server machines instead of from the multitude of clients. Another advantage could be **security**; having all servers in a locked machine room prevents certain types of problems from arising.

### CRUX: HOW TO BUILD A DISTRIBUTED FILE SYSTEM

How do you build a distributed file system? What are the key aspects to think about? What is easy to get wrong? What can we learn from existing systems?

## 48.1 A Basic Distributed File System

We now will study the architecture of a simplified distributed file system. A simple client/server distributed file system has more components than the file systems we have studied so far. On the client side, there are client applications which access files and directories through the **client-side file system**. A client application issues **system calls** to the client-side file system (such as `open()`, `read()`, `write()`, `close()`, `mkdir()`, etc.) in order to access files which are stored on the server. Thus, to client applications, the file system does not appear to be any different than a local (disk-based) file system, except perhaps for performance; in this way, distributed file systems provide **transparent** access to files, an obvious goal; after all, who would want to use a file system that required a different set of APIs or otherwise was a pain to use?

The role of the client-side file system is to execute the actions needed to service those system calls. For example, if the client issues a `read()` request, the client-side file system may send a message to the **server-side file system** (or, as it is commonly called, the **file server**) to read a particular block; the file server will then read the block from disk (or its own in-memory cache), and send a message back to the client with the requested data. The client-side file system will then copy the data into the user buffer supplied to the `read()` system call and thus the request will complete. Note that a subsequent `read()` of the same block on the client may be **cached** in client memory or on the client's disk even; in the best such case, no network traffic need be generated.

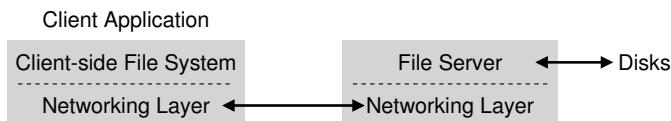


Figure 48.2: **Distributed File System Architecture**

From this simple overview, you should get a sense that there are two important pieces of software in a client/server distributed file system: the client-side file system and the file server. Together their behavior determines the behavior of the distributed file system. Now it's time to study one particular system: Sun's Network File System (NFS).

**ASIDE: WHY SERVERS CRASH**

Before getting into the details of the NFSv2 protocol, you might be wondering: why do servers crash? Well, as you might guess, there are plenty of reasons. Servers may simply suffer from a **power outage** (temporarily); only when power is restored can the machines be restarted. Servers are often comprised of hundreds of thousands or even millions of lines of code; thus, they have **bugs** (even good software has a few bugs per hundred or thousand lines of code), and thus they eventually will trigger a bug that will cause them to crash. They also have memory leaks; even a small memory leak will cause a system to run out of memory and crash. And, finally, in distributed systems, there is a network between the client and the server; if the network acts strangely (for example, if it becomes **partitioned** and clients and servers are working but cannot communicate), it may appear as if a remote machine has crashed, but in reality it is just not currently reachable through the network.

## 48.2 On To NFS

One of the earliest and quite successful distributed systems was developed by Sun Microsystems, and is known as the Sun Network File System (or NFS) [S86]. In defining NFS, Sun took an unusual approach: instead of building a proprietary and closed system, Sun instead developed an **open protocol** which simply specified the exact message formats that clients and servers would use to communicate. Different groups could develop their own NFS servers and thus compete in an NFS marketplace while preserving interoperability. It worked: today there are many companies that sell NFS servers (including Oracle/Sun, NetApp [HLM94], EMC, IBM, and others), and the widespread success of NFS is likely attributed to this “open market” approach.

## 48.3 Focus: Simple and Fast Server Crash Recovery

In this chapter, we will discuss the classic NFS protocol (version 2, a.k.a. NFSv2), which was the standard for many years; small changes were made in moving to NFSv3, and larger-scale protocol changes were made in moving to NFSv4. However, NFSv2 is both wonderful and frustrating and thus serves as our focus.

In NFSv2, the main goal in the design of the protocol was *simple and fast server crash recovery*. In a multiple-client, single-server environment, this goal makes a great deal of sense; any minute that the server is down (or unavailable) makes *all* the client machines (and their users) unhappy and unproductive. Thus, as the server goes, so goes the entire system.

## 48.4 Key To Fast Crash Recovery: Statelessness

This simple goal is realized in NFSv2 by designing what we refer to as a **stateless** protocol. The server, by design, does not keep track of anything about what is happening at each client. For example, the server does not know which clients are caching which blocks, or which files are currently open at each client, or the current file pointer position for a file, etc. Simply put, the server does not track anything about what clients are doing; rather, the protocol is designed to deliver in each protocol request *all the information* that is needed in order to complete the request. If it doesn't now, this stateless approach will make more sense as we discuss the protocol in more detail below.

For an example of a **stateful** (not stateless) protocol, consider the `open()` system call. Given a pathname, `open()` returns a file descriptor (an integer). This descriptor is used on subsequent `read()` or `write()` requests to access various file blocks, as in this application code (note that proper error checking of the system calls is omitted for space reasons):

```
char buffer[MAX];
int fd = open("foo", O_RDONLY); // get descriptor "fd"
read(fd, buffer, MAX);        // read MAX bytes from foo (via fd)
read(fd, buffer, MAX);        // read MAX bytes from foo
...
read(fd, buffer, MAX);        // read MAX bytes from foo
close(fd);                   // close file
```

Figure 48.3: Client Code: Reading From A File

Now imagine that the client-side file system opens the file by sending a protocol message to the server saying “open the file ‘foo’ and give me back a descriptor”. The file server then opens the file locally on its side and sends the descriptor back to the client. On subsequent reads, the client application uses that descriptor to call the `read()` system call; the client-side file system then passes the descriptor in a message to the file server, saying “read some bytes from the file that is referred to by the descriptor I am passing you here”.

In this example, the file descriptor is a piece of **shared state** between the client and the server (Ousterhout calls this **distributed state** [O91]). Shared state, as we hinted above, complicates crash recovery. Imagine the server crashes after the first read completes, but before the client has issued the second one. After the server is up and running again, the client then issues the second read. Unfortunately, the server has no idea to which file `fd` is referring; that information was ephemeral (i.e., in memory) and thus lost when the server crashed. To handle this situation, the client and server would have to engage in some kind of **recovery protocol**, where the client would make sure to keep enough information around in its memory to be able to tell the server what it needs to know (in this case, that file descriptor `fd` refers to file `foo`).

It gets even worse when you consider the fact that a stateful server has to deal with client crashes. Imagine, for example, a client that opens a file and then crashes. The `open()` uses up a file descriptor on the server; how can the server know it is OK to close a given file? In normal operation, a client would eventually call `close()` and thus inform the server that the file should be closed. However, when a client crashes, the server never receives a `close()`, and thus has to notice the client has crashed in order to close the file.

For these reasons, the designers of NFS decided to pursue a stateless approach: each client operation contains all the information needed to complete the request. No fancy crash recovery is needed; the server just starts running again, and a client, at worst, might have to retry a request.

## 48.5 The NFSv2 Protocol

We thus arrive at the NFSv2 protocol definition. Our problem statement is simple:

### THE CRUX: HOW TO DEFINE A STATELESS FILE PROTOCOL

How can we define the network protocol to enable stateless operation? Clearly, stateful calls like `open()` can't be a part of the discussion (as it would require the server to track open files); however, the client application will want to call `open()`, `read()`, `write()`, `close()` and other standard API calls to access files and directories. Thus, as a refined question, how do we define the protocol to both be stateless *and* support the POSIX file system API?

One key to understanding the design of the NFS protocol is understanding the **file handle**. File handles are used to uniquely describe the file or directory a particular operation is going to operate upon; thus, many of the protocol requests include a file handle.

You can think of a file handle as having three important components: a *volume identifier*, an *inode number*, and a *generation number*; together, these three items comprise a unique identifier for a file or directory that a client wishes to access. The volume identifier informs the server which file system the request refers to (an NFS server can export more than one file system); the inode number tells the server which file within that partition the request is accessing. Finally, the generation number is needed when reusing an inode number; by incrementing it whenever an inode number is reused, the server ensures that a client with an old file handle can't accidentally access the newly-allocated file.

Here is a summary of some of the important pieces of the protocol; the full protocol is available elsewhere (see Callaghan's book for an excellent and detailed overview of NFS [C00]).

```

NFSPROC_GETATTR
    expects: file handle
    returns: attributes
NFSPROC_SETATTR
    expects: file handle, attributes
    returns: nothing
NFSPROC_LOOKUP
    expects: directory file handle, name of file/directory to look up
    returns: file handle
NFSPROC_READ
    expects: file handle, offset, count
    returns: data, attributes
NFSPROC_WRITE
    expects: file handle, offset, count, data
    returns: attributes
NFSPROC_CREATE
    expects: directory file handle, name of file, attributes
    returns: nothing
NFSPROC_REMOVE
    expects: directory file handle, name of file to be removed
    returns: nothing
NFSPROC_MKDIR
    expects: directory file handle, name of directory, attributes
    returns: file handle
NFSPROC_RMDIR
    expects: directory file handle, name of directory to be removed
    returns: nothing
NFSPROC_READDIR
    expects: directory handle, count of bytes to read, cookie
    returns: directory entries, cookie (to get more entries)

```

#### Figure 48.4: The NFS Protocol: Examples

We briefly highlight the important components of the protocol. First, the LOOKUP protocol message is used to obtain a file handle, which is then subsequently used to access file data. The client passes a directory file handle and name of a file to look up, and the handle to that file (or directory) plus its attributes are passed back to the client from the server.

For example, assume the client already has a directory file handle for the root directory of a file system (/) (indeed, this would be obtained through the NFS **mount protocol**, which is how clients and servers first are connected together; we do not discuss the mount protocol here for sake of brevity). If an application running on the client opens the file /foo.txt, the client-side file system sends a lookup request to the server, passing it the root file handle and the name foo.txt; if successful, the file handle (and attributes) for foo.txt will be returned.

In case you are wondering, attributes are just the metadata that the file system tracks about each file, including fields such as file creation time, last modification time, size, ownership and permissions information, and so forth, i.e., the same type of information that you would get back if you called `stat()` on a file.

Once a file handle is available, the client can issue READ and WRITE protocol messages on a file to read or write the file, respectively. The READ protocol message requires the protocol to pass along the file handle

of the file along with the offset within the file and number of bytes to read. The server then will be able to issue the read (after all, the handle tells the server which volume and which inode to read from, and the offset and count tells it which bytes of the file to read) and return the data to the client (or an error if there was a failure). WRITE is handled similarly, except the data is passed from the client to the server, and just a success code is returned.

One last interesting protocol message is the GETATTR request; given a file handle, it simply fetches the attributes for that file, including the last modified time of the file. We will see why this protocol request is important in NFSv2 below when we discuss caching (can you guess why?).

## 48.6 From Protocol to Distributed File System

Hopefully you are now getting some sense of how this protocol is turned into a file system across the client-side file system and the file server. The client-side file system tracks open files, and generally translates application requests into the relevant set of protocol messages. The server simply responds to each protocol message, each of which has all the information needed to complete request.

For example, let us consider a simple application which reads a file. In the diagram (Figure 48.5), we show what system calls the application makes, and what the client-side file system and file server do in responding to such calls.

A few comments about the figure. First, notice how the client tracks all relevant **state** for the file access, including the mapping of the integer file descriptor to an NFS file handle as well as the current file pointer. This enables the client to turn each read request (which you may have noticed do *not* specify the offset to read from explicitly) into a properly-formatted read protocol message which tells the server exactly which bytes from the file to read. Upon a successful read, the client updates the current file position; subsequent reads are issued with the same file handle but a different offset.

Second, you may notice where server interactions occur. When the file is opened for the first time, the client-side file system sends a LOOKUP request message. Indeed, if a long pathname must be traversed (e.g., `/home/remzi/foo.txt`), the client would send three LOOKUPS: one to look up `home` in the directory `/`, one to look up `remzi` in `home`, and finally one to look up `foo.txt` in `remzi`.

Third, you may notice how each server request has all the information needed to complete the request in its entirety. This design point is critical to be able to gracefully recover from server failure, as we will now discuss in more detail; it ensures that the server does not need state to be able to respond to the request.

Client	Server
<pre><b>fd = open("/foo", ...);</b> Send LOOKUP (rootdir FH, "foo")</pre>	<p>Receive LOOKUP request look for "foo" in root dir return foo's FH + attributes</p>
<p>Receive LOOKUP reply allocate file desc in open file table store foo's FH in table store current file position (0) return file descriptor to application</p>	
<hr/> <pre><b>read(fd, buffer, MAX);</b> Index into open file table with fd get NFS file handle (FH) use current file position as offset Send READ (FH, offset=0, count=MAX)</pre>	<p>Receive READ request use FH to get volume/inode num read inode from disk (or cache) compute block location (using offset) read data from disk (or cache) return data to client</p>
<p>Receive READ reply update file position (+bytes read) set current file position = MAX return data/error code to app</p>	
<hr/> <pre><b>read(fd, buffer, MAX);</b> Same except offset=MAX and set current file position = 2*MAX</pre>	
<hr/> <pre><b>read(fd, buffer, MAX);</b> Same except offset=2*MAX and set current file position = 3*MAX</pre>	
<hr/> <pre><b>close(fd);</b> Just need to clean up local structures Free descriptor "fd" in open file table (No need to talk to server)</pre>	

Figure 48.5: Reading A File: Client-side And File Server Actions

**TIP: IDEMPOTENCY IS POWERFUL**

**Idempotency** is a useful property when building reliable systems. When an operation can be issued more than once, it is much easier to handle failure of the operation; you can just retry it. If an operation is *not* idempotent, life becomes more difficult.

## 48.7 Handling Server Failure with Idempotent Operations

When a client sends a message to the server, it sometimes does not receive a reply. There are many possible reasons for this failure to respond. In some cases, the message may be dropped by the network; networks do lose messages, and thus either the request or the reply could be lost and thus the client would never receive a response.

It is also possible that the server has crashed, and thus is not currently responding to messages. After a bit, the server will be rebooted and start running again, but in the meanwhile all requests have been lost. In all of these cases, clients are left with a question: what should they do when the server does not reply in a timely manner?

In NFSv2, a client handles all of these failures in a single, uniform, and elegant way: it simply *retries* the request. Specifically, after sending the request, the client sets a timer to go off after a specified time period. If a reply is received before the timer goes off, the timer is canceled and all is well. If, however, the timer goes off *before* any reply is received, the client assumes the request has not been processed and resends it. If the server replies, all is well and the client has neatly handled the problem.

The ability of the client to simply retry the request (regardless of what caused the failure) is due to an important property of most NFS requests: they are **idempotent**. An operation is called idempotent when the effect of performing the operation multiple times is equivalent to the effect of performing the operating a single time. For example, if you store a value to a memory location three times, it is the same as doing so once; thus “store value to memory” is an idempotent operation. If, however, you increment a counter three times, it results in a different amount than doing so just once; thus, “increment counter” is not idempotent. More generally, any operation that just reads data is obviously idempotent; an operation that updates data must be more carefully considered to determine if it has this property.

The heart of the design of crash recovery in NFS is the idempotency of most common operations. LOOKUP and READ requests are trivially idempotent, as they only read information from the file server and do not update it. More interestingly, WRITE requests are also idempotent. If, for example, a WRITE fails, the client can simply retry it. The WRITE message contains the data, the count, and (importantly) the exact offset to write the data to. Thus, it can be repeated with the knowledge that the outcome of multiple writes is the same as the outcome of a single one.

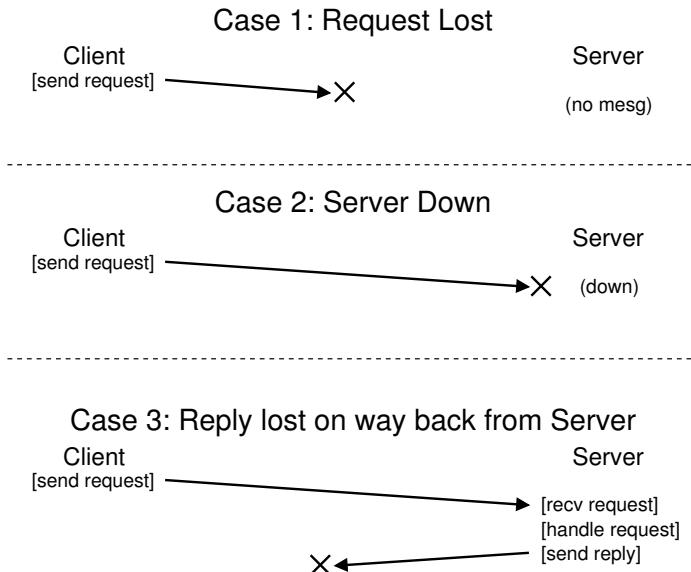


Figure 48.6: The Three Types of Loss

In this way, the client can handle all timeouts in a unified way. If a WRITE request was simply lost (Case 1 above), the client will retry it, the server will perform the write, and all will be well. The same will happen if the server happened to be down while the request was sent, but back up and running when the second request is sent, and again all works as desired (Case 2). Finally, the server may in fact receive the WRITE request, issue the write to its disk, and send a reply. This reply may get lost (Case 3), again causing the client to re-send the request. When the server receives the request again, it will simply do the exact same thing: write the data to disk and reply that it has done so. If the client this time receives the reply, all is again well, and thus the client has handled both message loss and server failure in a uniform manner. Neat!

A small aside: some operations are hard to make idempotent. For example, when you try to make a directory that already exists, you are informed that the mkdir request has failed. Thus, in NFS, if the file server receives a MKDIR protocol message and executes it successfully but the reply is lost, the client may repeat it and encounter that failure when in fact the operation at first succeeded and then only failed on the retry. Thus, life is not perfect.

**TIP: PERFECT IS THE ENEMY OF THE GOOD (VOLTAIRE'S LAW)**

Even when you design a beautiful system, sometimes all the corner cases don't work out exactly as you might like. Take the mkdir example above; one could redesign mkdir to have different semantics, thus making it idempotent (think about how you might do so); however, why bother? The NFS design philosophy covers most of the important cases, and overall makes the system design clean and simple with regards to failure. Thus, accepting that life isn't perfect and still building the system is a sign of good engineering. Apparently, this wisdom is attributed to Voltaire, for saying "... a wise Italian says that the best is the enemy of the good" [V72], and thus we call it **Voltaire's Law**.

## 48.8 Improving Performance: Client-side Caching

Distributed file systems are good for a number of reasons, but sending all read and write requests across the network can lead to a big performance problem: the network generally isn't that fast, especially as compared to local memory or disk. Thus, another problem: how can we improve the performance of a distributed file system?

The answer, as you might guess from reading the big bold words in the sub-heading above, is **client-side caching**. The NFS client-side file system caches file data (and metadata) that it has read from the server in client memory. Thus, while the first access is expensive (i.e., it requires network communication), subsequent accesses are serviced quite quickly out of client memory.

The cache also serves as a temporary buffer for writes. When a client application first writes to a file, the client buffers the data in client memory (in the same cache as the data it read from the file server) before writing the data out to the server. Such **write buffering** is useful because it decouples application `write()` latency from actual write performance, i.e., the application's call to `write()` succeeds immediately (and just puts the data in the client-side file system's cache); only later does the data get written out to the file server.

Thus, NFS clients cache data and performance is usually great and we are done, right? Unfortunately, not quite. Adding caching into any sort of system with multiple client caches introduces a big and interesting challenge which we will refer to as the **cache consistency problem**.

## 48.9 The Cache Consistency Problem

The cache consistency problem is best illustrated with two clients and a single server. Imagine client C1 reads a file F, and keeps a copy of the file in its local cache. Now imagine a different client, C2, overwrites the file F, thus changing its contents; let's call the new version of the file F

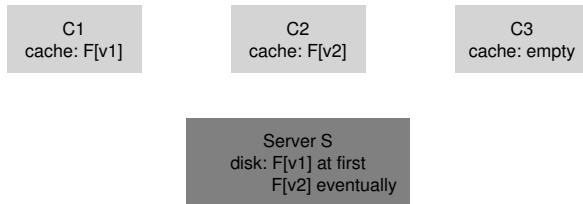


Figure 48.7: The Cache Consistency Problem

(version 2), or  $F[v2]$  and the old version  $F[v1]$  so we can keep the two distinct (but of course the file has the same name, just different contents). Finally, there is a third client,  $C_3$ , which has not yet accessed the file  $F$ .

You can probably see the problem that is upcoming (Figure 48.7). In fact, there are two subproblems. The first subproblem is that the client  $C_2$  may buffer its writes in its cache for a time before propagating them to the server; in this case, while  $F[v2]$  sits in  $C_2$ 's memory, any access of  $F$  from another client (say  $C_3$ ) will fetch the old version of the file ( $F[v1]$ ). Thus, by buffering writes at the client, other clients may get stale versions of the file, which may be undesirable; indeed, imagine the case where you log into machine  $C_2$ , update  $F$ , and then log into  $C_3$  and try to read the file, only to get the old copy! Certainly this could be frustrating. Thus, let us call this aspect of the cache consistency problem **update visibility**; when do updates from one client become visible at other clients?

The second subproblem of cache consistency is a **stale cache**; in this case,  $C_2$  has finally flushed its writes to the file server, and thus the server has the latest version ( $F[v2]$ ). However,  $C_1$  still has  $F[v1]$  in its cache; if a program running on  $C_1$  reads file  $F$ , it will get a stale version ( $F[v1]$ ) and not the most recent copy ( $F[v2]$ ), which is (often) undesirable.

NFSv2 implementations solve these cache consistency problems in two ways. First, to address update visibility, clients implement what is sometimes called **flush-on-close** (a.k.a., **close-to-open**) consistency semantics; specifically, when a file is written to and subsequently closed by a client application, the client flushes all updates (i.e., dirty pages in the cache) to the server. With flush-on-close consistency, NFS ensures that a subsequent open from another node will see the latest file version.

Second, to address the stale-cache problem, NFSv2 clients first check to see whether a file has changed before using its cached contents. Specifically, when opening a file, the client-side file system will issue a GETATTR request to the server to fetch the file's attributes. The attributes, importantly, include information as to when the file was last modified on the server; if the time-of-modification is more recent than the time that the file was fetched into the client cache, the client **invalidates** the file, thus removing it from the client cache and ensuring that subsequent reads will go to the server and retrieve the latest version of the file. If, on the other

hand, the client sees that it has the latest version of the file, it will go ahead and use the cached contents, thus increasing performance.

When the original team at Sun implemented this solution to the stale-cache problem, they realized a new problem; suddenly, the NFS server was flooded with GETATTR requests. A good engineering principle to follow is to design for the **common case**, and to make it work well; here, although the common case was that a file was accessed only from a single client (perhaps repeatedly), the client always had to send GETATTR requests to the server to make sure no one else had changed the file. A client thus bombards the server, constantly asking "has anyone changed this file?", when most of the time no one had.

To remedy this situation (somewhat), an **attribute cache** was added to each client. A client would still validate a file before accessing it, but most often would just look in the attribute cache to fetch the attributes. The attributes for a particular file were placed in the cache when the file was first accessed, and then would timeout after a certain amount of time (say 3 seconds). Thus, during those three seconds, all file accesses would determine that it was OK to use the cached file and thus do so with no network communication with the server.

## 48.10 Assessing NFS Cache Consistency

A few final words about NFS cache consistency. The flush-on-close behavior was added to "make sense", but introduced a certain performance problem. Specifically, if a temporary or short-lived file was created on a client and then soon deleted, it would still be forced to the server. A more ideal implementation might keep such short-lived files in memory until they are deleted and thus remove the server interaction entirely, perhaps increasing performance.

More importantly, the addition of an attribute cache into NFS made it very hard to understand or reason about exactly what version of a file one was getting. Sometimes you would get the latest version; sometimes you would get an old version simply because your attribute cache hadn't yet timed out and thus the client was happy to give you what was in client memory. Although this was fine most of the time, it would (and still does!) occasionally lead to odd behavior.

And thus we have described the oddity that is NFS client caching. It serves as an interesting example where details of an implementation serve to define user-observable semantics, instead of the other way around.

## 48.11 Implications on Server-Side Write Buffering

Our focus so far has been on client caching, and that is where most of the interesting issues arise. However, NFS servers tend to be well-equipped machines with a lot of memory too, and thus they have caching concerns as well. When data (and metadata) is read from disk, NFS

servers will keep it in memory, and subsequent reads of said data (and metadata) will not go to disk, a potential (small) boost in performance.

More intriguing is the case of write buffering. NFS servers absolutely may *not* return success on a WRITE protocol request until the write has been forced to stable storage (e.g., to disk or some other persistent device). While they can place a copy of the data in server memory, returning success to the client on a WRITE protocol request could result in incorrect behavior; can you figure out why?

The answer lies in our assumptions about how clients handle server failure. Imagine the following sequence of writes as issued by a client:

```
write(fd, a_buffer, size); // fill first block with a's
write(fd, b_buffer, size); // fill second block with b's
write(fd, c_buffer, size); // fill third block with c's
```

These writes overwrite the three blocks of a file with a block of a's, then b's, and then c's. Thus, if the file initially looked like this:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
```

We might expect the final result after these writes to be like this, with the x's, y's, and z's, would be overwritten with a's, b's, and c's, respectively.

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
cccccccccccccccccccccccccccccccccccccccccccccccccc
```

Now let's assume for the sake of the example that these three client writes were issued to the server as three distinct WRITE protocol messages. Assume the first WRITE message is received by the server and issued to the disk, and the client informed of its success. Now assume the second write is just buffered in memory, and the server also reports it success to the client *before* forcing it to disk; unfortunately, the server crashes before writing it to disk. The server quickly restarts and receives the third write request, which also succeeds.

Thus, to the client, all the requests succeeded, but we are surprised that the file contents look like this:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy <-- oops
cccccccccccccccccccccccccccccccccccccccccccccccccc
```

Yikes! Because the server told the client that the second write was successful before committing it to disk, an old chunk is left in the file, which, depending on the application, might be catastrophic.

To avoid this problem, NFS servers *must* commit each write to stable (persistent) storage before informing the client of success; doing so enables the client to detect server failure during a write, and thus retry until

it finally succeeds. Doing so ensures we will never end up with file contents intermingled as in the above example.

The problem that this requirement gives rise to in NFS server implementation is that write performance, without great care, can be *the* major performance bottleneck. Indeed, some companies (e.g., Network Appliance) came into existence with the simple objective of building an NFS server that can perform writes quickly; one trick they use is to first put writes in a battery-backed memory, thus enabling to quickly reply to WRITE requests without fear of losing the data and without the cost of having to write to disk right away; the second trick is to use a file system design specifically designed to write to disk quickly when one finally needs to do so [HLM94, RO91].

## 48.12 Summary

We have seen the introduction of the NFS distributed file system. NFS is centered around the idea of simple and fast recovery in the face of server failure, and achieves this end through careful protocol design. Idempotency of operations is essential; because a client can safely replay a failed operation, it is OK to do so whether or not the server has executed the request.

We also have seen how the introduction of caching into a multiple-client, single-server system can complicate things. In particular, the system must resolve the cache consistency problem in order to behave reasonably; however, NFS does so in a slightly ad hoc fashion which can occasionally result in observably weird behavior. Finally, we saw how server caching can be tricky: writes to the server must be forced to stable storage before returning success (otherwise data can be lost).

We haven't talked about other issues which are certainly relevant, notably security. Security in early NFS implementations was remarkably lax; it was rather easy for any user on a client to masquerade as other users and thus gain access to virtually any file. Subsequent integration with more serious authentication services (e.g., Kerberos [NT94]) have addressed these obvious deficiencies.

## References

- [C00] "NFS Illustrated"  
 Brent Callaghan  
 Addison-Wesley Professional Computing Series, 2000  
*A great NFS reference; incredibly thorough and detailed per the protocol itself.*
- [HLM94] "File System Design for an NFS File Server Appliance"  
 Dave Hitz, James Lau, Michael Malcolm  
 USENIX Winter 1994. San Francisco, California, 1994  
*Hitz et al. were greatly influenced by previous work on log-structured file systems.*
- [NT94] "Kerberos: An Authentication Service for Computer Networks"  
 B. Clifford Neuman, Theodore Ts'o  
 IEEE Communications, 32(9):33-38, September 1994  
*Kerberos is an early and hugely influential authentication service. We probably should write a book chapter about it sometime...*
- [O91] "The Role of Distributed State"  
 John K. Ousterhout  
 Available: <ftp://ftp.cs.berkeley.edu/ucb/sprite/papers/state.ps>  
*A rarely referenced discussion of distributed state; a broader perspective on the problems and challenges.*
- [P+94] "NFS Version 3: Design and Implementation"  
 Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, Dave Hitz  
 USENIX Summer 1994, pages 137-152  
*The small modifications that underlie NFS version 3.*
- [P+00] "The NFS version 4 protocol"  
 Brian Pawlowski, David Noveck, David Robinson, Robert Thurlow  
 2nd International System Administration and Networking Conference (SANE 2000)  
*Undoubtedly the most literary paper on NFS ever written.*
- [RO91] "The Design and Implementation of the Log-structured File System"  
 Mendel Rosenblum, John Ousterhout  
 Symposium on Operating Systems Principles (SOSP), 1991  
*LFS again. No, you can never get enough LFS.*
- [S86] "The Sun Network File System: Design, Implementation and Experience"  
 Russel Sandberg  
 USENIX Summer 1986  
*The original NFS paper; though a bit of a challenging read, it is worthwhile to see the source of these wonderful ideas.*
- [Sun89] "NFS: Network File System Protocol Specification"  
 Sun Microsystems, Inc. Request for Comments: 1094, March 1989  
 Available: <http://www.ietf.org/rfc/rfc1094.txt>  
*The dreaded specification; read it if you must, i.e., you are getting paid to read it. Hopefully, paid a lot. Cash money!*
- [V72] "La Begueule"  
 Francois-Marie Arouet a.k.a. Voltaire  
 Published in 1772  
*Voltaire said a number of clever things, this being but one example. For example, Voltaire also said "If you have two religions in your land, the two will cut each others throats; but if you have thirty religions, they will dwell in peace." What do you say to that, Democrats and Republicans?*

## The Andrew File System (AFS)

The Andrew File System was introduced by researchers at Carnegie-Mellon University (CMU) in the 1980's [H+88]. Led by the well-known Professor M. Satyanarayanan of Carnegie-Mellon University ("Satya" for short), the main goal of this project was simple: **scale**. Specifically, how can one design a distributed file system such that a server can support as many clients as possible?

Interestingly, there are numerous aspects of design and implementation that affect scalability. Most important is the design of the **protocol** between clients and servers. In NFS, for example, the protocol forces clients to check with the server periodically to determine if cached contents have changed; because each check uses server resources (including CPU and network bandwidth), frequent checks like this will limit the number of clients a server can respond to and thus limit scalability.

AFS also differs from NFS in that from the beginning, reasonable user-visible behavior was a first-class concern. In NFS, cache consistency is hard to describe because it depends directly on low-level implementation details, including client-side cache timeout intervals. In AFS, cache consistency is simple and readily understood: when the file is opened, a client will generally receive the latest consistent copy from the server.

### 49.1 AFS Version 1

We will discuss two versions of AFS [H+88, S+85]. The first version (which we will call AFSv1, but actually the original system was called the ITC distributed file system [S+85]) had some of the basic design in place, but didn't scale as desired, which led to a re-design and the final protocol (which we will call AFSv2, or just AFS) [H+88]. We now discuss the first version.

One of the basic tenets of all versions of AFS is **whole-file caching** on the **local disk** of the client machine that is accessing a file. When you `open()` a file, the entire file (if it exists) is fetched from the server and stored in a file on your local disk. Subsequent application `read()` and `write()` operations are redirected to the local file system where the file is

TestAuth	Test whether a file has changed (used to validate cached entries)
GetFileStat	Get the stat info for a file
Fetch	Fetch the contents of file
Store	Store this file on the server
SetFileStat	Set the stat info for a file
ListDir	List the contents of a directory

Figure 49.1: AFSv1 Protocol Highlights

stored; thus, these operations require no network communication and are fast. Finally, upon `close()`, the file (if it has been modified) is flushed back to the server. Note the obvious contrasts with NFS, which caches *blocks* (not whole files, although NFS could of course cache every block of an entire file) and does so in client *memory* (not local disk).

Let's get into the details a bit more. When a client application first calls `open()`, the AFS client-side code (which the AFS designers call **Venus**) would send a Fetch protocol message to the server. The Fetch protocol message would pass the entire pathname of the desired file (for example, `/home/remzi/notes.txt`) to the file server (the group of which they called **Vice**), which would then traverse the pathname, find the desired file, and ship the entire file back to the client. The client-side code would then cache the file on the local disk of the client (by writing it to local disk). As we said above, subsequent `read()` and `write()` system calls are strictly *local* in AFS (no communication with the server occurs); they are just redirected to the local copy of the file. Because the `read()` and `write()` calls act just like calls to a local file system, once a block is accessed, it also may be cached in client memory. Thus, AFS also uses client memory to cache copies of blocks that it has in its local disk. Finally, when finished, the AFS client checks if the file has been modified (i.e., that it has been opened for writing); if so, it flushes the new version back to the server with a Store protocol message, sending the entire file and pathname to the server for permanent storage.

The next time the file is accessed, AFSv1 does so much more efficiently. Specifically, the client-side code first contacts the server (using the TestAuth protocol message) in order to determine whether the file has changed. If not, the client would use the locally-cached copy, thus improving performance by avoiding a network transfer. The figure above shows some of the protocol messages in AFSv1. Note that this early version of the protocol only cached file contents; directories, for example, were only kept at the server.

## 49.2 Problems with Version 1

A few key problems with this first version of AFS motivated the designers to rethink their file system. To study the problems in detail, the designers of AFS spent a great deal of time measuring their existing prototype to find what was wrong. Such experimentation is a good thing;

**TIP: MEASURE THEN BUILD (PATTERSON'S LAW)**

One of our advisors, David Patterson (of RISC and RAID fame), used to always encourage us to measure a system and demonstrate a problem *before* building a new system to fix said problem. By using experimental evidence, rather than gut instinct, you can turn the process of system building into a more scientific endeavor. Doing so also has the fringe benefit of making you think about how exactly to measure the system before your improved version is developed. When you do finally get around to building the new system, two things are better as a result: first, you have evidence that shows you are solving a real problem; second, you now have a way to measure your new system in place, to show that it actually improves upon the state of the art. And thus we call this **Patterson's Law**.

**measurement** is the key to understanding how systems work and how to improve them. Hard data helps take intuition and make into a concrete science of deconstructing systems. In their study, the authors found two main problems with AFSv1:

- **Path-traversal costs are too high:** When performing a Fetch or Store protocol request, the client passes the entire pathname (e.g., /home/remzi/notes.txt) to the server. The server, in order to access the file, must perform a full pathname traversal, first looking in the root directory to find home, then in home to find remzi, and so forth, all the way down the path until finally the desired file is located. With many clients accessing the server at once, the designers of AFS found that the server was spending much of its CPU time simply walking down directory paths.
- **The client issues too many TestAuth protocol messages:** Much like NFS and its overabundance of GETATTR protocol messages, AFSv1 generated a large amount of traffic to check whether a local file (or its stat information) was valid with the TestAuth protocol message. Thus, servers spent much of their time telling clients whether it was OK to use their cached copies of a file. Most of the time, the answer was that the file had not changed.

There were actually two other problems with AFSv1: load was not balanced across servers, and the server used a single distinct process per client thus inducing context switching and other overheads. The load imbalance problem was solved by introducing **volumes**, which an administrator could move across servers to balance load; the context-switch problem was solved in AFSv2 by building the server with threads instead of processes. However, for the sake of space, we focus here on the main two protocol problems above that limited the scale of the system.

### 49.3 Improving the Protocol

The two problems above limited the scalability of AFS; the server CPU became the bottleneck of the system, and each server could only service 20 clients without becoming overloaded. Servers were receiving too many TestAuth messages, and when they received Fetch or Store messages, were spending too much time traversing the directory hierarchy. Thus, the AFS designers were faced with a problem:

#### THE CRUX: HOW TO DESIGN A SCALABLE FILE PROTOCOL

How should one redesign the protocol to minimize the number of server interactions, i.e., how could they reduce the number of TestAuth messages? Further, how could they design the protocol to make these server interactions efficient? By attacking both of these issues, a new protocol would result in a much more scalable version AFS.

### 49.4 AFS Version 2

AFSv2 introduced the notion of a **callback** to reduce the number of client/server interactions. A callback is simply a promise from the server to the client that the server will inform the client when a file that the client is caching has been modified. By adding this **state** to the server, the client no longer needs to contact the server to find out if a cached file is still valid. Rather, it assumes that the file is valid until the server tells it otherwise; insert analogy to **polling** versus **interrupts** here.

AFSv2 also introduced the notion of a **file identifier (FID)** (similar to the NFS **file handle**) instead of pathnames to specify which file a client was interested in. An FID in AFS consists of a volume identifier, a file identifier, and a “uniquifier” (to enable reuse of the volume and file IDs when a file is deleted). Thus, instead of sending whole pathnames to the server and letting the server walk the pathname to find the desired file, the client would walk the pathname, one piece at a time, caching the results and thus hopefully reducing the load on the server.

For example, if a client accessed the file `/home/remzi/notes.txt`, and `home` was the AFS directory mounted onto `/` (i.e., `/` was the local root directory, but `home` and its children were in AFS), the client would first Fetch the directory contents of `home`, put them in the local-disk cache, and setup a callback on `home`. Then, the client would Fetch the directory `remzi`, put it in the local-disk cache, and setup a callback on the server on `remzi`. Finally, the client would Fetch `notes.txt`, cache this regular file in the local disk, setup a callback, and finally return a file descriptor to the calling application. See Figure 49.2 for a summary.

The key difference, however, from NFS, is that with each fetch of a directory or file, the AFS client would establish a callback with the server, thus ensuring that the server would notify the client of a change in its

Client (C <sub>1</sub> )	Server
<pre>fd = open("/home/remzi/notes.txt", ...); Send Fetch (home FID, "remzi")</pre>	<pre>Receive Fetch request look for remzi in home dir establish callback(C<sub>1</sub>) on remzi return remzi's content and FID</pre>
<pre>Receive Fetch reply write remzi to local disk cache record callback status of remzi Send Fetch (remzi FID, "notes.txt")</pre>	<pre>Receive Fetch request look for notes.txt in remzi dir establish callback(C<sub>1</sub>) on notes.txt return notes.txt's content and FID</pre>
<pre>Receive Fetch reply write notes.txt to local disk cache record callback status of notes.txt local open() of cached notes.txt return file descriptor to application</pre>	
<hr/> <pre>read(fd, buffer, MAX); perform local read() on cached copy</pre>	
<hr/> <pre>close(fd); do local close() on cached copy if file has changed, flush to server</pre>	
<hr/> <pre>fd = open("/home/remzi/notes.txt", ...); Foreach dir (home, remzi) if (callback(dir) == VALID) use local copy for lookup(dir) else Fetch (as above) if (callback(notes.txt) == VALID) open local cached copy return file descriptor to it else Fetch (as above) then open and return fd</pre>	

Figure 49.2: Reading A File: Client-side And File Server Actions

cached state. The benefit is obvious: although the *first* access to `/home/remzi/notes.txt` generates many client-server messages (as described above), it also establishes callbacks for all the directories as well as the file `notes.txt`, and thus subsequent accesses are entirely local and require no server interaction at all. Thus, in the common case where a file is cached at the client, AFS behaves nearly identically to a local disk-based file system. If one accesses a file more than once, the second access should be just as fast as accessing a file locally.

**ASIDE: CACHE CONSISTENCY IS NOT A PANACEA**

When discussing distributed file systems, much is made of the cache consistency the file systems provide. However, this baseline consistency does not solve all problems with regards to file access from multiple clients. For example, if you are building a code repository, with multiple clients performing check-ins and check-outs of code, you can't simply rely on the underlying file system to do all of the work for you; rather, you have to use explicit **file-level locking** in order to ensure that the “right” thing happens when such concurrent accesses take place. Indeed, any application that truly cares about concurrent updates will add extra machinery to handle conflicts. The baseline consistency described in this chapter and the previous one are useful primarily for casual usage, i.e., when a user logs into a different client, they expect some reasonable version of their files to show up there. Expecting more from these protocols is setting yourself up for failure, disappointment, and tear-filled frustration.

## 49.5 Cache Consistency

When we discussed NFS, there were two aspects of cache consistency we considered: **update visibility** and **cache staleness**. With update visibility, the question is: when will the server be updated with a new version of a file? With cache staleness, the question is: once the server has a new version, how long before clients see the new version instead of an older cached copy?

Because of callbacks and whole-file caching, the cache consistency provided by AFS is easy to describe and understand. There are two important cases to consider: consistency between processes on *different* machines, and consistency between processes on the *same* machine.

Between different machines, AFS makes updates visible at the server and invalidates cached copies at the exact same time, which is when the updated file is closed. A client opens a file, and then writes to it (perhaps repeatedly). When it is finally closed, the new file is flushed to the server (and thus visible); the server then breaks callbacks for any clients with cached copies, thus ensuring that clients will no longer read stale copies of the file; subsequent opens on those clients will require a re-fetch of the new version of the file from the server.

AFS makes an exception to this simple model between processes on the same machine. In this case, writes to a file are immediately visible to other local processes (i.e., a process does not have to wait until a file is closed to see its latest updates). This makes using a single machine behave exactly as you would expect, as this behavior is based upon typical UNIX semantics. Only when switching to a different machine would you be able to detect the more general AFS consistency mechanism.

There is one interesting cross-machine case that is worthy of further discussion. Specifically, in the rare case that processes on different ma-

$P_1$	$P_2$	$P_3$	$P_1$	$P_2$	$P_3$	Server Disk	Comments
Cache	Cache	Cache	Cache	Cache	Cache	Disk	
open(F)	-		-			-	File created
write(A)	A		-			-	
close()	A		-			A	
	open(F)	A	-			A	
	read() → A	A	-			A	
	close()	A	-			A	
open(F)	A		-			A	
write(B)	B		-			A	
	open(F)	B	-			A	Local processes
	read() → B	B	-			A	see writes immediately
	close()	B	-			A	
		B	open(F)	A	A	A	Remote processes
		B	read() → A	A	A	A	do not see writes...
		B	close()	A	A	A	
close()	B			A	B	B	... until close()
		B	open(F)	B	B	B	has taken place
		B	read() → B	B	B	B	
		B	close()	B	B	B	
		B	open(F)	B	B	B	
open(F)	B		-			B	
write(D)	D		-			B	
		D	write(C)	C	C	B	
		D	close()	C	C	C	
close()	D			A	D	D	Unfortunately for $P_3$
		D	open(F)	D	D	D	the last writer wins
		D	read() → D	D	D	D	
		D	close()	D	D	D	

Figure 49.3: Cache Consistency Timeline

achines are modifying a file at the same time, AFS naturally employs what is known as a **last writer wins** approach (which perhaps should be called **last closer wins**). Specifically, whichever client calls `close()` last will update the entire file on the server last and thus will be the “winning” file, i.e., the file that remains on the server for others to see. The result is a file that was generated in its entirety either by one client or the other. Note the difference from a block-based protocol like NFS: in NFS, writes of individual blocks may be flushed out to the server as each client is updating the file, and thus the final file on the server could end up as a mix of updates from both clients. In many cases, such a mixed file output would not make much sense, i.e., imagine a JPEG image getting modified by two clients in pieces; the resulting mix of writes would not likely constitute a valid JPEG.

A timeline showing a few of these different scenarios can be seen in Figure 49.3. The columns show the behavior of two processes ( $P_1$  and  $P_2$ ) on Client<sub>1</sub> and its cache state, one process ( $P_3$ ) on Client<sub>2</sub> and its cache state, and the server (Server), all operating on a single file called, imaginatively, F. For the server, the figure simply shows the contents of the file after the operation on the left has completed. Read through it and see if you can understand why each read returns the results that it does. A commentary field on the right will help you if you get stuck.

## 49.6 Crash Recovery

From the description above, you might sense that crash recovery is more involved than with NFS. You would be right. For example, imagine there is a short period of time where a server (S) is not able to contact a client (C1), for example, while the client C1 is rebooting. While C1 is not available, S may have tried to send it one or more callback recall messages; for example, imagine C1 had file F cached on its local disk, and then C2 (another client) updated F, thus causing S to send messages to all clients caching the file to remove it from their local caches. Because C1 may miss those critical messages when it is rebooting, upon rejoining the system, C1 should treat all of its cache contents as suspect. Thus, upon the next access to file F, C1 should first ask the server (with a `TestAuth` protocol message) whether its cached copy of file F is still valid; if so, C1 can use it; if not, C1 should fetch the newer version from the server.

Server recovery after a crash is also more complicated. The problem that arises is that callbacks are kept in memory; thus, when a server reboots, it has no idea which client machine has which files. Thus, upon server restart, each client of the server must realize that the server has crashed and treat all of their cache contents as suspect, and (as above) reestablish the validity of a file before using it. Thus, a server crash is a big event, as one must ensure that each client is aware of the crash in a timely manner, or risk a client accessing a stale file. There are many ways to implement such recovery; for example, by having the server send a message (saying “don’t trust your cache contents!”) to each client when it is up and running again, or by having clients check that the server is alive periodically (with a `heartbeat` message, as it is called). As you can see, there is a cost to building a more scalable and sensible caching model; with NFS, clients hardly noticed a server crash.

## 49.7 Scale And Performance Of AFSv2

With the new protocol in place, AFSv2 was measured and found to be much more scalable than the original version. Indeed, each server could support about 50 clients (instead of just 20). A further benefit was that client-side performance often came quite close to local performance, because in the common case, all file accesses were local; file reads usually went to the local disk cache (and potentially, local memory). Only when a client created a new file or wrote to an existing one was there need to send a `Store` message to the server and thus update the file with new contents.

Let us also gain some perspective on AFS performance by comparing common file-system access scenarios with NFS. Figure 49.4 (page 9) shows the results of our qualitative comparison.

In the figure, we examine typical read and write patterns analytically, for files of different sizes. Small files have  $N_s$  blocks in them; medium files have  $N_m$  blocks; large files have  $N_L$  blocks. We assume that small

Workload	NFS	AFS	AFS/NFS
1. Small file, sequential read	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
2. Small file, sequential re-read	$N_s \cdot L_{mem}$	$N_s \cdot L_{mem}$	1
3. Medium file, sequential read	$N_m \cdot L_{net}$	$N_m \cdot L_{net}$	1
4. Medium file, sequential re-read	$N_m \cdot L_{mem}$	$N_m \cdot L_{mem}$	1
5. Large file, sequential read	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
6. Large file, sequential re-read	$N_L \cdot L_{net}$	$N_L \cdot L_{disk}$	$\frac{L_{disk}}{L_{net}}$
7. Large file, single read	$L_{net}$	$N_L \cdot L_{net}$	$\frac{N_L}{L_{net}}$
8. Small file, sequential write	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
9. Large file, sequential write	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
10. Large file, sequential overwrite	$N_L \cdot L_{net}$	$2 \cdot N_L \cdot L_{net}$	2
11. Large file, single write	$L_{net}$	$2 \cdot N_L \cdot L_{net}$	$2 \cdot N_L$

Figure 49.4: Comparison: AFS vs. NFS

and medium files fit into the memory of a client; large files fit on a local disk but not in client memory.

We also assume, for the sake of analysis, that an access across the network to the remote server for a file block takes  $L_{net}$  time units. Access to local memory takes  $L_{mem}$ , and access to local disk takes  $L_{disk}$ . The general assumption is that  $L_{net} > L_{disk} > L_{mem}$ .

Finally, we assume that the first access to a file does not hit in any caches. Subsequent file accesses (i.e., “re-reads”) we assume will hit in caches, if the relevant cache has enough capacity to hold the file.

The columns of the figure show the time a particular operation (e.g., a small file sequential read) roughly takes on either NFS or AFS. The right-most column displays the ratio of AFS to NFS.

We make the following observations. First, in many cases, the performance of each system is roughly equivalent. For example, when first reading a file (e.g., Workloads 1, 3, 5), the time to fetch the file from the remote server dominates, and is similar on both systems. You might think AFS would be slower in this case, as it has to write the file to local disk; however, those writes are buffered by the local (client-side) file system cache and thus said costs are likely hidden. Similarly, you might think that AFS reads from the local cached copy would be slower, again because AFS stores the cached copy on disk. However, AFS again benefits here from local file system caching; reads on AFS would likely hit in the client-side memory cache, and performance would be similar to NFS.

Second, an interesting difference arises during a large-file sequential re-read (Workload 6). Because AFS has a large local disk cache, it will access the file from there when the file is accessed again. NFS, in contrast, only can cache blocks in client memory; as a result, if a large file (i.e., a file bigger than local memory) is re-read, the NFS client will have to re-fetch the entire file from the remote server. Thus, AFS is faster than NFS in this case by a factor of  $\frac{L_{net}}{L_{disk}}$ , assuming that remote access is indeed slower than local disk. We also note that NFS in this case increases server load, which has an impact on scale as well.

Third, we note that sequential writes (of new files) should perform similarly on both systems (Workloads 8, 9). AFS, in this case, will write the file to the local cached copy; when the file is closed, the AFS client will force the writes to the server, as per the protocol. NFS will buffer writes in client memory, perhaps forcing some blocks to the server due to client-side memory pressure, but definitely writing them to the server when the file is closed, to preserve NFS flush-on-close consistency. You might think AFS would be slower here, because it writes all data to local disk. However, realize that it is writing to a local file system; those writes are first committed to the page cache, and only later (in the background) to disk, and thus AFS reaps the benefits of the client-side OS memory caching infrastructure to improve performance.

Fourth, we note that AFS performs worse on a sequential file overwrite (Workload 10). Thus far, we have assumed that the workloads that write are also creating a new file; in this case, the file exists, and is then over-written. Overwrite can be a particularly bad case for AFS, because the client first fetches the old file in its entirety, only to subsequently overwrite it. NFS, in contrast, will simply overwrite blocks and thus avoid the initial (useless) read<sup>1</sup>.

Finally, workloads that access a small subset of data within large files perform much better on NFS than AFS (Workloads 7, 11). In these cases, the AFS protocol fetches the entire file when the file is opened; unfortunately, only a small read or write is performed. Even worse, if the file is modified, the entire file is written back to the server, doubling the performance impact. NFS, as a block-based protocol, performs I/O that is proportional to the size of the read or write.

Overall, we see that NFS and AFS make different assumptions and not surprisingly realize different performance outcomes as a result. Whether these differences matter is, as always, a question of workload.

## 49.8 AFS: Other Improvements

Like we saw with the introduction of Berkeley FFS (which added symbolic links and a number of other features), the designers of AFS took the opportunity when building their system to add a number of features that made the system easier to use and manage. For example, AFS provides a true global namespace to clients, thus ensuring that all files were named the same way on all client machines. NFS, in contrast, allows each client to mount NFS servers in any way that they please, and thus only by convention (and great administrative effort) would files be named similarly across clients.

---

<sup>1</sup>We assume here that NFS reads are block-sized and block-aligned; if they were not, the NFS client would also have to read the block first. We also assume the file was *not* opened with the O\_TRUNC flag; if it had been, the initial open in AFS would not fetch the soon to be truncated file's contents.

**ASIDE: THE IMPORTANCE OF WORKLOAD**

One challenge of evaluating any system is the choice of **workload**. Because computer systems are used in so many different ways, there are a large variety of workloads to choose from. How should the storage system designer decide which workloads are important, in order to make reasonable design decisions?

The designers of AFS, given their experience in measuring how file systems were used, made certain workload assumptions; in particular, they assumed that most files were not frequently shared, and accessed sequentially in their entirety. Given those assumptions, the AFS design makes perfect sense.

However, these assumptions are not always correct. For example, imagine an application that appends information, periodically, to a log. These little log writes, which add small amounts of data to an existing large file, are quite problematic for AFS. Many other difficult workloads exist as well, e.g., random updates in a transaction database.

One place to get some information about what types of workloads are common are through various research studies that have been performed. See any of these studies for good examples of workload analysis [B+91, H+11, R+00, V99], including the AFS retrospective [H+88].

AFS also takes security seriously, and incorporates mechanisms to authenticate users and ensure that a set of files could be kept private if a user so desired. NFS, in contrast, had quite primitive support for security for many years.

AFS also includes facilities for flexible user-managed access control. Thus, when using AFS, a user has a great deal of control over who exactly can access which files. NFS, like most UNIX file systems, has much less support for this type of sharing.

Finally, as mentioned before, AFS adds tools to enable simpler management of servers for the administrators of the system. In thinking about system management, AFS was light years ahead of the field.

## 49.9 Summary

AFS shows us how distributed file systems can be built quite differently than what we saw with NFS. The protocol design of AFS is particularly important; by minimizing server interactions (through whole-file caching and callbacks), each server can support many clients and thus reduce the number of servers needed to manage a particular site. Many other features, including the single namespace, security, and access-control lists, make AFS quite nice to use. The consistency model provided by AFS is simple to understand and reason about, and does not lead to the occasional weird behavior as one sometimes observes in NFS.

Perhaps unfortunately, AFS is likely on the decline. Because NFS became an open standard, many different vendors supported it, and, along with CIFS (the Windows-based distributed file system protocol), NFS dominates the marketplace. Although one still sees AFS installations from time to time (such as in various educational institutions, including Wisconsin), the only lasting influence will likely be from the ideas of AFS rather than the actual system itself. Indeed, NFSv4 now adds server state (e.g., an “open” protocol message), and thus bears an increasing similarity to the basic AFS protocol.

## References

[B+91] "Measurements of a Distributed File System"

Mary Baker, John Hartman, Martin Kupfer, Ken Shirriff, John Ousterhout  
SOSP '91, Pacific Grove, California, October 1991

*An early paper measuring how people use distributed file systems. Matches much of the intuition found in AFS.*

[H+11] "A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications"

Tyler Harter, Chris Dragga, Michael Vaughn,  
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau  
SOSP '11, New York, New York, October 2011

*Our own paper studying the behavior of Apple Desktop workloads; turns out they are a bit different than many of the server-based workloads the systems research community usually focuses upon. Also a good recent reference which points to a lot of related work.*

[H+88] "Scale and Performance in a Distributed File System"

John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan,  
Robert N. Sidebotham, Michael J. West  
ACM Transactions on Computing Systems (ACM TOCS), page 51-81, Volume 6, Number 1,  
February 1988

*The long journal version of the famous AFS system, still in use in a number of places throughout the world, and also probably the earliest clear thinking on how to build distributed file systems. A wonderful combination of the science of measurement and principled engineering.*

[R+00] "A Comparison of File System Workloads"

Drew Roselli, Jacob R. Lorch, Thomas E. Anderson  
USENIX '00, San Diego, California, June 2000

*A more recent set of traces as compared to the Baker paper [B+91], with some interesting twists.*

[S+85] "The ITC Distributed File System: Principles and Design"

M. Satyanarayanan, J.H. Howard, D.A. Nichols, R.N. Sidebotham, A. Spector, M.J. West  
SOSP '85, Orcas Island, Washington, December 1985

*The older paper about a distributed file system. Much of the basic design of AFS is in place in this older system, but not the improvements for scale.*

[V99] "File system usage in Windows NT 4.0"

Werner Vogels  
SOSP '99, Kiawah Island Resort, South Carolina, December 1999

*A cool study of Windows workloads, which are inherently different than many of the UNIX-based studies that had previously been done.*

## Homework

This section introduces `afs.py`, a simple AFS simulator you can use to shore up your knowledge of how the Andrew File System works. Read the README file for more details.

### Questions

1. Run a few simple cases to make sure you can predict what values will be read by clients. Vary the random seed flag (`-s`) and see if you can trace through and predict both intermediate values as well as the final values stored in the files. Also vary the number of files (`-f`), the number of clients (`-c`), and the read ratio (`-r`, from between 0 to 1) to make it a bit more challenging. You might also want to generate slightly longer traces to make for more interesting interactions, e.g., (`-n 2` or higher).
2. Now do the same thing and see if you can predict each callback that the AFS server initiates. Try different random seeds, and make sure to use a high level of detailed feedback (e.g., `-d 3`) to see when callbacks occur when you have the program compute the answers for you (with `-c`). Can you guess exactly when each callback occurs? What is the precise condition for one to take place?
3. Similar to above, run with some different random seeds and see if you can predict the exact cache state at each step. Cache state can be observed by running with `-c` and `-d 7`.
4. Now let's construct some specific workloads. Run the simulation with `-A oal:w1:c1,oal:r1:c1` flag. What are different possible values observed by client 1 when it reads the file `a`, when running with the random scheduler? (try different random seeds to see different outcomes)? Of all the possible schedule interleavings of the two clients' operations, how many of them lead to client 1 reading the value 1, and how many reading the value 0?
5. Now let's construct some specific schedules. When running with the `-A oal:w1:c1,oal:r1:c1` flag, also run with the following schedules: `-S 01`, `-S 100011`, `-S 011100`, and others of which you can think. What value will client 1 read?
6. Now run with this workload: `-A oal:w1:c1,oal:w1:c1`, and vary the schedules as above. What happens when you run with `-S 011100`? What about when you run with `-S 010011`? What is important in determining the final value of the file?

- 
- 
- 
- 
- 
- *EN*
- 
- 
- +
- 

## ACID



**Connected to:**

[Database](#) [Andreas Reuter](#) [Theo Härdter](#)

## From Wikipedia, the free encyclopedia

In [computer science](#), **ACID** (*Atomicity*, *Consistency*, *Isolation*, *Durability*) is a set of properties of [database transactions](#). In the context of [databases](#), a single logical operation on the data is called a transaction. For example, a transfer of funds from one bank account to another, even involving multiple changes such as debiting one account and crediting another, is a single transaction.

[Jim Gray](#) defined these properties of a reliable transaction system in the late 1970s and developed technologies to achieve them automatically.[\[1\]](#)[\[2\]](#)[\[3\]](#)

In 1983, [Andreas Reuter](#) and [Theo Härdter](#) coined the acronym *ACID* to describe them.[\[4\]](#)

## Characteristics

The characteristics of these four properties as defined by Reuter and Härdter:

### Atomicity

Main article: [Atomicity \(database systems\)](#)

[Atomicity](#) requires that each transaction be "all or nothing": if one part of the transaction fails, then the entire transaction fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes. To the outside world, a committed transaction appears (by its effects on the database) to be indivisible ("atomic"), and an aborted transaction does not happen.

### Consistency

Main article: [Consistency \(database systems\)](#)

The [consistency](#) property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including [constraints](#), [cascades](#), [triggers](#), and any combination thereof. This does not guarantee correctness of the transaction in all ways the application programmer might have wanted (that is the responsibility of application-level code) but merely that any programming errors cannot result in the violation of any defined rules.

## Isolation

Main article: [Isolation \(database systems\)](#)

The [isolation](#) property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e., one after the other. Providing isolation is the main goal of [concurrency control](#). Depending on the concurrency control method (i.e., if it uses strict - as opposed to [relaxed](#) - serializability), the effects of an incomplete transaction might not even be visible to another transaction.

## Durability

Main article: [Durability \(database systems\)](#)

The [durability](#) property ensures that once a transaction has been committed, it will remain so, even in the event of power loss, [crashes](#), or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter). To defend against power loss, transactions (or their effects) must be recorded in a [non-volatile memory](#).

## Examples

The following examples further illustrate the ACID properties. In these examples, the database table has two columns, A and B. An [integrity constraint](#) requires that the value in A and the value in B must sum to 100. The following [SQL](#) code creates a table as described above:

```
CREATE TABLE acidtest (A INTEGER, B INTEGER, CHECK (A + B = 100));
```

## Atomicity failure

This section does not cite any sources. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. (March 2015) (Learn how and when to remove this template message)

In database systems, atomicity (or atomicness; from Greek a-tomos, undividable) is one of the ACID transaction properties. In an atomic transaction, a series of database operations either all occur, or nothing occurs. The series of operations cannot be divided apart and executed partially from each other, which makes the series of operations "indivisible", hence the name. A guarantee of atomicity prevents updates to the database occurring only partially, which can cause greater problems than rejecting the whole series outright. In other words, atomicity means indivisibility and irreducibility.

## Consistency failure

This section does not cite any sources. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. (March 2015) (Learn how and when to remove this template message)

Consistency is a very general term, which demands that the data must meet all validation rules. In the previous example, the validation is a requirement that  $A + B = 100$ . Also, it may be inferred that both A and B must be integers. A valid range for A and B may also be inferred. All validation rules must be checked to ensure consistency. Assume that a transaction attempts to subtract 10 from A without altering B. Because consistency is checked after each transaction, it is known that  $A + B = 100$  before the transaction begins. If the transaction removes 10 from A successfully, atomicity will be achieved. However, a validation check will show that  $A + B = 90$ , which is inconsistent with the rules of the database. The entire transaction must be cancelled and the affected rows rolled back to their pre-transaction state. If there had been other constraints, triggers, or cascades, every single change operation would have been checked in the same way as above before the transaction was committed.

## Isolation failure

This section does not cite any sources. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. (March 2015) (Learn how and when to remove this template message)

To demonstrate isolation, we assume two transactions execute at the same time, each attempting to modify the same data. One of the two must wait until the other completes in order to maintain isolation.

Consider two transactions.  $T_1$  transfers 10 from A to B.  $T_2$  transfers 10 from B to A. Combined, there are four actions:

- $T_1$  subtracts 10 from A.
- $T_1$  adds 10 to B.
- $T_2$  subtracts 10 from B.
- $T_2$  adds 10 to A.

If these operations are performed in order, isolation is maintained, although  $T_2$  must wait. Consider what happens if  $T_1$  fails halfway through. The database eliminates  $T_1$ 's effects, and  $T_2$  sees only valid data.

By interleaving the transactions, the actual order of actions might be:

- $T_1$  subtracts 10 from A.
- $T_2$  subtracts 10 from B.
- $T_2$  adds 10 to A.
- $T_1$  adds 10 to B.

Again, consider what happens if  $T_1$  fails halfway through. By the time  $T_1$  fails,  $T_2$  has already modified A; it cannot be restored to the value it had before  $T_1$  without leaving an invalid database. This is known as a [write-write failure](#), [\[citation needed\]](#) because two transactions attempted to write to the same data field. In a typical system, the problem would be resolved by reverting to the last known good state, canceling the failed transaction  $T_1$ , and restarting the interrupted transaction  $T_2$  from the good state.

## Durability failure

This section does not cite any sources. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. (March 2015) (Learn how and when to remove this template message)

Consider a transaction that transfers 10 from A to B. First it removes 10 from A, then it adds 10 to B. At this point, the user is told the transaction was a success, however the changes are still queued in the [disk buffer](#) waiting to be committed to disk. Power fails and the changes are lost. The user assumes (understandably) that the changes have been persisted.

## Implementation

This section does not cite any sources. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. (March 2015) (Learn how and when to remove this template message)

Processing a transaction often requires a sequence of operations that is subject to failure for a number of reasons. For instance, the system may have no room left on its disk drives, or it may have used up its allocated CPU time. There are two popular families of techniques: [write-ahead logging](#) and [shadow paging](#). In both cases, [locks](#) must be acquired on all information to be updated, and depending on the level of isolation, possibly on all data that be read as well. In write ahead logging, atomicity is guaranteed by copying the original (unchanged) data to a log before changing the database.<sup>[[dubious](#) – [discuss](#)]</sup> That allows the database to return to a consistent state in the event of a crash. In shadowing, updates are applied to a partial copy of the database, and the new copy is activated when the transaction commits.

## Locking vs multiversioning

This section does not cite any sources. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. (March 2015) (Learn how and when to remove this template message)

Many databases rely upon locking to provide ACID capabilities. Locking means that the transaction marks the data that it accesses so that the DBMS knows not to allow other transactions to modify it until the first transaction succeeds or fails. The lock must always be acquired before processing data, including data that is read but not modified. Non-trivial transactions typically require a large number of locks, resulting in substantial overhead as well as blocking other transactions. For example, if user A is running a transaction that has to read a row of data that user B wants to modify, user B must wait until user A's transaction completes. [Two phase locking](#) is often applied to guarantee full isolation.

An alternative to locking is [multiversion concurrency control](#), in which the database provides each reading transaction the prior, unmodified version of data that is being modified by another active transaction. This allows readers to operate without acquiring locks, i.e. writing transactions do not block reading transactions, and readers do not block writers. Going back to the example, when user A's transaction requests data that user B is modifying, the database provides A with the version of that data that existed when user B started his transaction. User A gets a consistent view of the database even if other users are changing data. One implementation, namely [snapshot isolation](#), relaxes the isolation property.

## Distributed transactions

This section does not cite any sources. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. (March 2015) (Learn how and when to remove this template message)

Guaranteeing ACID properties in a [distributed transaction](#) across a [distributed database](#), where no single node is responsible for all data affecting a transaction, presents additional complications. Network connections might fail, or one node might successfully complete its part of the transaction and then be required to roll back its changes because of a failure on another node. The [two-phase commit protocol](#) (not to be confused with [two-phase locking](#)) provides atomicity for [distributed transactions](#) to ensure that each participant in the transaction agrees on whether the transaction should be committed or not. [citation needed] Briefly, in the first phase, one node (the coordinator) interrogates the other nodes (the participants) and only when all reply that they are prepared does the coordinator, in the second phase, formalize the transaction.

## See also

- [Basically Available, Soft state, Eventual consistency](#) (BASE)
- [CAP theorem](#)
- [Concurrency control](#)
- [Java Transaction API](#)
- [Open Systems Interconnection](#)
- [Transactional NTFS](#)
- [Two-phase commit protocol](#)
- [CRUD](#)

## References

1. [^ "Gray to be Honored With A.M. Turing Award This Spring"](#). Microsoft PressPass. [Archived](#) from the original on February 6, 2009. Retrieved March 27, 2015.
2. [^ Gray, Jim](#) (September 1981). ["The Transaction Concept: Virtues and Limitations"](#) (PDF). Proceedings of the 7th International Conference on Very Large Databases. Cupertino, CA: [Tandem Computers](#). pp. 144–154. Retrieved March 27, 2015.
3. [^ Gray, Jim & Andreas Reuter](#). *Distributed Transaction Processing: Concepts and Techniques*. [Morgan Kaufmann](#), 1993; [ISBN 1-55860-190-2](#).
4. [^ Haerder, T.; Reuter, A.](#) (1983). ["Principles of transaction-oriented database recovery"](#). *ACM Computing Surveys*. **15** (4): 287. [doi:10.1145/289.291](#). These four properties, atomicity, consistency, isolation, and durability (ACID), describe the major highlights of the transaction paradigm, which has influenced many aspects of development in database systems.

[\[hide\]](#)

y  
t  
e

## [Database management systems](#)

Types

- [Object-oriented](#)
  - [comparison](#)
- [Relational](#)
  - [comparison](#)
- [Document-oriented](#)
- [Graph](#)
- [NoSQL](#)
- [NewSQL](#)

## Virtual Machine Monitors

### B.1 Introduction

Years ago, IBM sold expensive mainframes to large organizations, and a problem arose: what if the organization wanted to run different operating systems on the machine at the same time? Some applications had been developed on one OS, and some on others, and thus the problem. As a solution, IBM introduced yet another level of indirection in the form of a **virtual machine monitor (VMM)** (also called a **hypervisor**) [G74].

Specifically, the monitor sits between one or more operating systems and the hardware and gives the illusion to each running OS that it controls the machine. Behind the scenes, however, the monitor actually is in control of the hardware, and must multiplex running OSes across the physical resources of the machine. Indeed, the VMM serves as an operating system for operating systems, but at a much lower level; the OS must still think it is interacting with the physical hardware. Thus, **transparency** is a major goal of VMMs.

Thus, we find ourselves in a funny position: the OS has thus far served as the master illusionist, tricking unsuspecting applications into thinking they have their own private CPU and a large virtual memory, while secretly switching between applications and sharing memory as well. Now, we have to do it again, but this time underneath the OS, who is used to being in charge. How can the VMM create this illusion for each OS running on top of it?

#### THE CRUX:

#### HOW TO VIRTUALIZE THE MACHINE UNDERNEATH THE OS

The virtual machine monitor must transparently virtualize the machine underneath the OS; what are the techniques required to do so?

## B.2 Motivation: Why VMMs?

Today, VMMs have become popular again for a multitude of reasons. Server consolidation is one such reason. In many settings, people run services on different machines which run different operating systems (or even OS versions), and yet each machine is lightly utilized. In this case, virtualization enables an administrator to **consolidate** multiple OSes onto fewer hardware platforms, and thus lower costs and ease administration.

Virtualization has also become popular on desktops, as many users wish to run one operating system (say Linux or Mac OS X) but still have access to native applications on a different platform (say Windows). This type of improvement in **functionality** is also a good reason.

Another reason is testing and debugging. While developers write code on one main platform, they often want to debug and test it on the many different platforms that they deploy the software to in the field. Thus, virtualization makes it easy to do so, by enabling a developer to run many operating system types and versions on just one machine.

This resurgence in virtualization began in earnest the mid-to-late 1990's, and was led by a group of researchers at Stanford headed by Professor Mendel Rosenblum. His group's work on Disco [B+97], a virtual machine monitor for the MIPS processor, was an early effort that revived VMMs and eventually led that group to the founding of VMware [V98], now a market leader in virtualization technology. In this chapter, we will discuss the primary technology underlying Disco and through that window try to understand how virtualization works.

## B.3 Virtualizing the CPU

To run a **virtual machine** (e.g., an OS and its applications) on top of a virtual machine monitor, the basic technique that is used is **limited direct execution**, a technique we saw before when discussing how the OS virtualizes the CPU. Thus, when we wish to "boot" a new OS on top of the VMM, we simply jump to the address of the first instruction and let the OS begin running. It is as simple as that (well, almost).

Assume we are running on a single processor, and that we wish to multiplex between two virtual machines, that is, between two OSes and their respective applications. In a manner quite similar to an operating system switching between running processes (a **context switch**), a virtual machine monitor must perform a **machine switch** between running virtual machines. Thus, when performing such a switch, the VMM must save the entire machine state of one OS (including registers, PC, and unlike in a context switch, any privileged hardware state), restore the machine state of the to-be-run VM, and then jump to the PC of the to-be-run VM and thus complete the switch. Note that the to-be-run VM's PC may be within the OS itself (i.e., the system was executing a system call) or it may simply be within a process that is running on that OS (i.e., a user-mode application).

We get into some slightly trickier issues when a running application or OS tries to perform some kind of **privileged operation**. For example, on a system with a software-managed TLB, the OS will use special privileged instructions to update the TLB with a translation before restarting an instruction that suffered a TLB miss. In a virtualized environment, the OS cannot be allowed to perform privileged instructions, because then it controls the machine rather than the VMM beneath it. Thus, the VMM must somehow intercept attempts to perform privileged operations and thus retain control of the machine.

A simple example of how a VMM must interpose on certain operations arises when a running process on a given OS tries to make a system call. For example, the process may be trying to call `open()` on a file, or may be calling `read()` to get data from it, or may be calling `fork()` to create a new process. In a system without virtualization, a system call is achieved with a special instruction; on MIPS, it is a `trap` instruction, and on x86, it is the `int` (an interrupt) instruction with the argument `0x80`. Here is the `open` library call on FreeBSD [B00] (recall that your C code first makes a library call into the C library, which then executes the proper assembly sequence to actually issue the trap instruction and make a system call):

```
open:  
    push    dword mode  
    push    dword flags  
    push    dword path  
    mov     eax, 5  
    push    eax  
    int    80h
```

On UNIX-based systems, `open()` takes just three arguments: `int open(char *path, int flags, mode_t mode)`. You can see in the code above how the `open()` library call is implemented: first, the arguments get pushed onto the stack (`mode`, `flags`, `path`), then a `5` gets pushed onto the stack, and then `int 80h` is called, which transfers control to the kernel. The `5`, if you were wondering, is the pre-agreed upon convention between user-mode applications and the kernel for the `open()` system call in FreeBSD; different system calls would place different numbers onto the stack (in the same position) before calling the trap instruction `int` and thus making the system call<sup>1</sup>.

When a trap instruction is executed, as we've discussed before, it usually does a number of interesting things. Most important in our example here is that it first transfers control (i.e., changes the PC) to a well-defined **trap handler** within the operating system. The OS, when it is first starting up, establishes the address of such a routine with the hardware (also a privileged operation) and thus upon subsequent traps, the hardware

---

<sup>1</sup>Just to make things confusing, the Intel folks use the term "interrupt" for what almost any sane person would call a trap instruction. As Patterson said about the Intel instruction set: "It's an ISA only a mother could love." But actually, we kind of like it, and we're not its mother.

Process	Hardware	Operating System
1. Execute instructions (add, load, etc.)		
2. System call: Trap to OS	3. Switch to kernel mode; Jump to trap handler	4. In kernel mode; Handle system call; Return from trap
	5. Switch to user mode; Return to user code	
6. Resume execution (@PC after trap)		

Table B.1: Executing a System Call

knows where to start running code to handle the trap. At the same time of the trap, the hardware also does one other crucial thing: it changes the mode of the processor from **user mode** to **kernel mode**. In user mode, operations are restricted, and attempts to perform privileged operations will lead to a trap and likely the termination of the offending process; in kernel mode, on the other hand, the full power of the machine is available, and thus all privileged operations can be executed. Thus, in a traditional setting (again, without virtualization), the flow of control would be like what you see in Table B.1.

On a virtualized platform, things are a little more interesting. When an application running on an OS wishes to perform a system call, it does the exact same thing: executes a trap instruction with the arguments carefully placed on the stack (or in registers). However, it is the VMM that controls the machine, and thus the VMM who has installed a trap handler that will first get executed in kernel mode.

So what should the VMM do to handle this system call? The VMM doesn't really know **how** to handle the call; after all, it does not know the details of each OS that is running and therefore does not know what each call should do. What the VMM does know, however, is **where** the OS's trap handler is. It knows this because when the OS booted up, it tried to install its own trap handlers; when the OS did so, it was trying to do something privileged, and therefore trapped into the VMM; at that time, the VMM recorded the necessary information (i.e., where this OS's trap handlers are in memory). Now, when the VMM receives a trap from a user process running on the given OS, it knows exactly what to do: it jumps to the OS's trap handler and lets the OS handle the system call as it should. When the OS is finished, it executes some kind of privileged instruction to return from the trap (`rett` on MIPS, `iret` on x86), which again bounces into the VMM, which then realizes that the OS is trying to return from the trap and thus performs a real return-from-trap and thus returns control to the user and puts the machine back in user mode. The entire process is depicted in Tables B.2 and B.3, both for the normal case without virtualization and the case with virtualization (we leave out the exact hardware operations from above to save space).

Process	Operating System
1. System call: Trap to OS	2. OS trap handler: Decode trap and execute appropriate syscall routine; When done: return from trap
3. Resume execution (@PC after trap)	

Table B.2: **System Call Flow Without Virtualization**

Process	Operating System	VMM
1. System call: Trap to OS	2. Process trapped: Call OS trap handler (at reduced privilege)	
3. OS trap handler: Decode trap and execute syscall; When done: issue return-from-trap	4. OS tried return from trap: Do real return from trap	
5. Resume execution (@PC after trap)		

Table B.3: **System Call Flow with Virtualization**

As you can see from the figures, a lot more has to take place when virtualization is going on. Certainly, because of the extra jumping around, virtualization might indeed slow down system calls and thus could hurt performance.

You might also notice that we have one remaining question: what mode should the OS run in? It can't run in kernel mode, because then it would have unrestricted access to the hardware. Thus, it must run in some less privileged mode than before, be able to access its own data structures, and simultaneously prevent access to its data structures from user processes.

In the Disco work, Rosenblum and colleagues handled this problem quite neatly by taking advantage of a special mode provided by the MIPS hardware known as supervisor mode. When running in this mode, one still doesn't have access to privileged instructions, but one can access a little more memory than when in user mode; the OS can use this extra memory for its data structures and all is well. On hardware that doesn't have such a mode, one has to run the OS in user mode and use memory protection (page tables and TLBs) to protect OS data structures appropriately. In other words, when switching into the OS, the monitor would have to make the memory of the OS data structures available to the OS via page-table protections; when switching back to the running application, the ability to read and write the kernel would have to be removed.

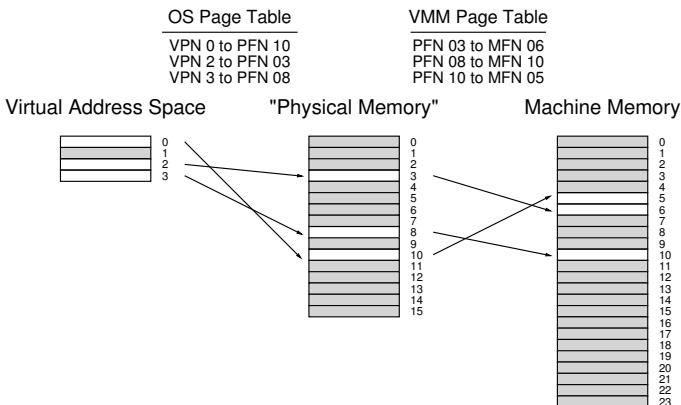


Figure B.1: VMM Memory Virtualization

## B.4 Virtualizing Memory

You should now have a basic idea of how the processor is virtualized: the VMM acts like an OS and schedules different virtual machines to run, and some interesting interactions occur when privilege levels change. But we have left out a big part of the equation: how does the VMM virtualize memory?

Each OS normally thinks of physical memory as a linear array of pages, and assigns each page to itself or user processes. The OS itself, of course, already virtualizes memory for its running processes, such that each process has the illusion of its own private address space. Now we must add another layer of virtualization, so that multiple OSes can share the actual physical memory of the machine, and we must do so transparently.

This extra layer of virtualization makes “physical” memory a virtualization on top of what the VMM refers to as **machine memory**, which is the real physical memory of the system. Thus, we now have an additional layer of indirection: each OS maps virtual-to-physical addresses via its per-process page tables; the VMM maps the resulting physical mappings to underlying machine addresses via its per-OS page tables. Figure B.1 depicts this extra level of indirection.

In the figure, there is just a single virtual address space with four pages, three of which are valid (0, 2, and 3). The OS uses its page table to map these pages to three underlying physical frames (10, 3, and 8, respectively). Underneath the OS, the VMM performs a further level of indirection, mapping PFNs 3, 8, and 10 to machine frames 6, 10, and 5 respectively. Of course, this picture simplifies things quite a bit; on a real system, there would be  $V$  operating systems running (with  $V$  likely

Process	Operating System
1. Load from memory: TLB miss: Trap	2. OS TLB miss handler: Extract VPN from VA; Do page table lookup; If present and valid: get PFN, update TLB; Return from trap
3. Resume execution (@PC of trapping instruction); Instruction is retried; Results in TLB hit	

Table B.4: TLB Miss Flow without Virtualization

greater than one), and thus  $V$  VMM page tables; further, on top of each running operating system  $OS_i$ , there would be a number of processes  $P_i$  running ( $P_i$  likely in the tens or hundreds), and hence  $P_i$  (per-process) page tables within  $OS_i$ .

To understand how this works a little better, let's recall how **address translation** works in a modern paged system. Specifically, let's discuss what happens on a system with a software-managed TLB during address translation. Assume a user process generates an address (for an instruction fetch or an explicit load or store); by definition, the process generates a **virtual address**, as its address space has been virtualized by the OS. As you know by now, it is the role of the OS, with help from the hardware, to turn this into a **physical address** and thus be able to fetch the desired contents from physical memory.

Assume we have a 32-bit virtual address space and a 4-KB page size. Thus, our 32-bit address is chopped into two parts: a 20-bit virtual page number (VPN), and a 12-bit offset. The role of the OS, with help from the hardware TLB, is to translate the VPN into a valid physical page frame number (PFN) and thus produce a fully-formed physical address which can be sent to physical memory to fetch the proper data. In the common case, we expect the TLB to handle the translation in hardware, thus making the translation fast. When a TLB miss occurs (at least, on a system with a software-managed TLB), the OS must get involved to service the miss, as depicted here in Table B.4.

As you can see, a TLB miss causes a trap into the OS, which handles the fault by looking up the VPN in the page table and installing the translation in the TLB.

With a virtual machine monitor underneath the OS, however, things again get a little more interesting. Let's examine the flow of a TLB miss again (see Table B.5 for a summary). When a process makes a virtual memory reference and misses in the TLB, it is not the OS TLB miss handler that runs; rather, it is the VMM TLB miss handler, as the VMM is the true privileged owner of the machine. However, in the normal case, the VMM TLB handler doesn't know how to handle the TLB miss, so it immediately jumps into the OS TLB miss handler; the VMM knows the

Process	Operating System	Virtual Machine Monitor
1. Load from memory TLB miss: Trap		2. VMM TLB miss handler: Call into OS TLB handler (reducing privilege)
3. OS TLB miss handler: Extract VPN from VA; Do page table lookup; If present and valid, get PFN, update TLB		4. Trap handler: Unprivileged code trying to update the TLB; OS is trying to install VPN-to-PFN mapping; Update TLB instead with VPN-to-MFN (privileged); Jump back to OS (reducing privilege)
5. Return from trap		6. Trap handler: Unprivileged code trying to return from a trap; Return from trap
7. Resume execution (@PC of instruction); Instruction is retried; Results in TLB hit		

Table B.5: TLB Miss Flow with Virtualization

location of this handler because the OS, during “boot”, tried to install its own trap handlers. The OS TLB miss handler then runs, does a page table lookup for the VPN in question, and tries to install the VPN-to-PFN mapping in the TLB. However, doing so is a privileged operation, and thus causes another trap into the VMM (the VMM gets notified when any non-privileged code tries to do something that is privileged, of course). At this point, the VMM plays its trick: instead of installing the OS’s VPN-to-PFN mapping, the VMM installs its desired VPN-to-MFN mapping. After doing so, the system eventually gets back to the user-level code, which retries the instruction, and results in a TLB hit, fetching the data from the machine frame where the data resides.

This set of actions also hints at how a VMM must manage the virtualization of physical memory for each running OS; just like the OS has a page table for each process, the VMM must track the physical-to-machine mappings for each virtual machine it is running. These per-machine page tables need to be consulted in the VMM TLB miss handler in order to determine which machine page a particular “physical” page maps to, and even, for example, if it is present in machine memory at the current time (i.e., the VMM could have swapped it to disk).

**ASIDE: HYPERVERISORS AND HARDWARE-MANAGED TLBs**

Our discussion has centered around software-managed TLBs and the work that needs to be done when a miss occurs. But you might be wondering: how does the virtual machine monitor get involved with a hardware-managed TLB? In those systems, the hardware walks the page table on each TLB miss and updates the TLB as need be, and thus the VMM doesn't have a chance to run on each TLB miss to sneak its translation into the system. Instead, the VMM must closely monitor changes the OS makes to each page table (which, in a hardware-managed system, is pointed to by a page-table base register of some kind), and keep a **shadow page table** that instead maps the virtual addresses of each process to the VMM's desired machine pages [AA06]. The VMM installs a process's shadow page table whenever the OS tries to install the process's OS-level page table, and thus the hardware chugs along, translating virtual addresses to machine addresses using the shadow table, without the OS even noticing.

Finally, as you might notice from this sequence of operations, TLB misses on a virtualized system become quite a bit more expensive than in a non-virtualized system. To reduce this cost, the designers of Disco added a VMM-level "software TLB". The idea behind this data structure is simple. The VMM records every virtual-to-physical mapping that it sees the OS try to install; then, on a TLB miss, the VMM first consults its software TLB to see if it has seen this virtual-to-physical mapping before, and what the VMM's desired virtual-to-machine mapping should be. If the VMM finds the translation in its software TLB, it simply installs the virtual-to-machine mapping directly into the hardware TLB, and thus skips all the back and forth in the control flow above [B+97].

## B.5 The Information Gap

Just like the OS doesn't know too much about what application programs really want, and thus must often make general policies that hopefully work for all programs, the VMM often doesn't know too much about what the OS is doing or wanting; this lack of knowledge, sometimes called the **information gap** between the VMM and the OS, can lead to various inefficiencies [B+97]. For example, an OS, when it has nothing else to run, will sometimes go into an **idle loop** just spinning and waiting for the next interrupt to occur:

```
while (1)
    ; // the idle loop
```

It makes sense to spin like this if the OS in charge of the entire machine and thus knows there is nothing else that needs to run. However, when a

**ASIDE: PARA-VIRTUALIZATION**

In many situations, it is good to assume that the OS cannot be modified in order to work better with virtual machine monitors (for example, because you are running your VMM under an unfriendly competitor's operating system). However, this is not always the case, and when the OS can be modified (as we saw in the example with demand-zeroing of pages), it may run more efficiently on top of a VMM. Running a modified OS to run on a VMM is generally called **para-virtualization** [WSG02], as the virtualization provided by the VMM isn't a complete one, but rather a partial one requiring OS changes to operate effectively. Research shows that a properly-designed para-virtualized system, with just the right OS changes, can be made to be nearly as efficient a system without a VMM [BD+03].

VMM is running underneath two different OSes, one in the idle loop and one usefully running user processes, it would be useful for the VMM to know that one OS is idle so it can give more CPU time to the OS doing useful work.

Another example arises with demand zeroing of pages. Most operating systems zero a physical frame before mapping it into a process's address space. The reason for doing so is simple: security. If the OS gave one process a page that another had been using *without* zeroing it, an information leak across processes could occur, thus potentially leaking sensitive information. Unfortunately, the VMM must zero pages that it gives to each OS, for the same reason, and thus many times a page will be zeroed twice, once by the VMM when assigning it to an OS, and once by the OS when assigning it to a process. The authors of Disco had no great solution to this problem: they simply changed the OS (IRIX) to not zero pages that it knew had been zeroed by the underlying VMM [B+97].

There are many other similar problems to these described here. One solution is for the VMM to use inference (a form of **implicit information**) to overcome the problem. For example, a VMM can detect the idle loop by noticing that the OS switched to low-power mode. A different approach, seen in **para-virtualized** systems, requires the OS to be changed. This more explicit approach, while harder to deploy, can be quite effective.

## B.6 Summary

Virtualization is in a renaissance. For a multitude of reasons, users and administrators want to run multiple OSes on the same machine at the same time. The key is that VMMs generally provide this service **transparently**; the OS above has little clue that it is not actually controlling the hardware of the machine. The key method that VMMs use to do so is to extend the notion of limited direct execution; by setting up the hard-

**TIP: USE IMPLICIT INFORMATION**

Implicit information can be a powerful tool in layered systems where it is hard to change the interfaces between systems, but more information about a different layer of the system is needed. For example, a block-based disk device might like to know more about how a file system above it is using it; Similarly, an application might want to know what pages are currently in the file-system page cache, but the OS provides no API to access this information. In both these cases, researchers have developed powerful inferencing techniques to gather the needed information implicitly, *without* requiring an explicit interface between layers [AD+01,S+03]. Such techniques are quite useful in a virtual machine monitor, which would like to learn more about the OSes running above it without requiring an explicit API between the two layers.

ware to enable the VMM to interpose on key events (such as traps), the VMM can completely control how machine resources are allocated while preserving the illusion that the OS requires.

You might have noticed some similarities between what the OS does for processes and what the VMM does for OSes. They both virtualize the hardware after all, and hence do some of the same things. However, there is one key difference: with the OS virtualization, a number of new abstractions and nice interfaces are provided; with VMM-level virtualization, the abstraction is identical to the hardware (and thus not very nice). While both the OS and VMM virtualize hardware, they do so by providing completely different interfaces; VMMs, unlike the OS, are not particularly meant to make the hardware easier to use.

There are many other topics to study if you wish to learn more about virtualization. For example, we didn't even discuss what happens with I/O, a topic that has its own new and interesting issues when it comes to virtualized platforms. We also didn't discuss how virtualization works when running "on the side" with your OS in what is sometimes called a "hosted" configuration. Read more about both of these topics if you're interested [SVL01]. We also didn't discuss what happens when a collection of operating systems running on a VMM uses too much memory.

Finally, hardware support has changed how platforms support virtualization. Companies like Intel and AMD now include direct support for an extra level of virtualization, thus obviating many of the software techniques in this chapter. Perhaps, in a chapter yet-to-be-written, we will discuss these mechanisms in more detail.

## References

- [AA06] "A Comparison of Software and Hardware Techniques for x86 Virtualization"  
 Keith Adams and Ole Agesen  
 ASPLOS '06, San Jose, California  
*A terrific paper from two VMware engineers about the surprisingly small benefits of having hardware support for virtualization. Also an excellent general discussion about virtualization in VMware, including the crazy binary-translation tricks they have to play in order to virtualize the difficult-to-virtualize x86 platform.*
- [AD+01] "Information and Control in Gray-box Systems"  
 Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau  
 SOSP '01, Banff, Canada  
*Our own work on how to infer information and even exert control over the OS from application level, without any change to the OS. The best example therein: determining which file blocks are cached in the OS using a probabilistic probe-based technique; doing so allows applications to better utilize the cache, by first scheduling work that will result in hits.*
- [B00] "FreeBSD Developers' Handbook:  
 Chapter 11 x86 Assembly Language Programming"  
<http://www.freebsd.org/doc/en/books/developers-handbook/>  
*A nice tutorial on system calls and such in the BSD developers handbook.*
- [BD+03] "Xen and the Art of Virtualization"  
 Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield  
 SOSP '03, Bolton Landing, New York  
*The paper that shows that with para-virtualized systems, the overheads of virtualized systems can be made to be incredibly low. So successful was this paper on the Xen virtual machine monitor that it launched a company.*
- [B+97] "Disco: Running Commodity Operating Systems on Scalable Multiprocessors"  
 Edouard Bugnion, Scott Devine, Kinshuk Govil, Mendel Rosenblum  
 SOSP '97  
*The paper that reintroduced the systems community to virtual machine research; well, perhaps this is unfair as Bressoud and Schneider [BS95] also did, but here we began to understand why virtualization was going to come back. What made it even clearer, however, is when this group of excellent researchers started VMware and made some billions of dollars.*
- [BS95] "Hypervisor-based Fault-tolerance"  
 Thomas C. Bressoud, Fred B. Schneider  
 SOSP '95  
*One the earliest papers to bring back the **hypervisor**, which is just another term for a virtual machine monitor. In this work, however, such hypervisors are used to improve system tolerance of hardware faults, which is perhaps less useful than some of the more practical scenarios discussed in this chapter; however, still quite an intriguing paper in its own right.*
- [G74] "Survey of Virtual Machine Research"  
 R.P. Goldberg  
 IEEE Computer, Volume 7, Number 6  
*A terrific survey of a lot of old virtual machine research.*

[SVL01] "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor"

Jeremy Sugerman, Ganesh Venkitachalam and Beng-Hong Lim  
USENIX '01, Boston, Massachusetts

*Provides a good overview of how I/O works in VMware using a hosted architecture which exploits many native OS features to avoid reimplementing them within the VMM.*

[V98] VMware corporation.

Available: <http://www.vmware.com/>

*This may be the most useless reference in this book, as you can clearly look this up yourself. Anyhow, the company was founded in 1998 and is a leader in the field of virtualization.*

[S+03] "Semantically-Smart Disk Systems"

Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpacı-Dusseau, Remzi H. Arpacı-Dusseau  
FAST '03, San Francisco, California, March 2003

*Our work again, this time showing how a dumb block-based device can infer much about what the file system above it is doing, such as deleting a file. The technology used therein enables interesting new functionality within a block device, such as secure delete, or more reliable storage.*

[WSG02] "Scale and Performance in the Denali Isolation Kernel"

Andrew Whitaker, Marianne Shaw, and Steven D. Gribble  
OSDI '02, Boston, Massachusetts

*The paper that introduces the term para-virtualization. Although one can argue that Bugnion et al. [B+97] introduce the idea of para-virtualization in the Disco paper, Whitaker et al. take it further and show how the idea can be more general than what was thought before.*

## 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.

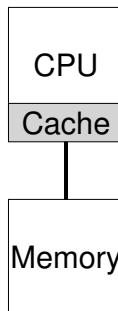


Figure 10.1: Single CPU With Cache

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.

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

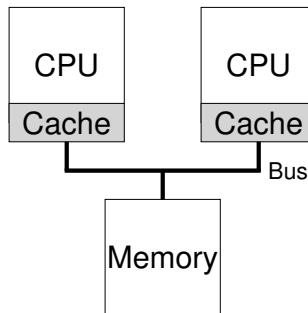


Figure 10.2: Two CPUs With Caches Sharing Memory

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

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

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

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

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

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

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

## 10.2 Don't Forget Synchronization

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

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

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

```

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

Figure 10.3: Simple List Delete Code

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

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

### 10.3 One Final Issue: Cache Affinity

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

### 10.4 Single-Queue Scheduling

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

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

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

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



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

CPU 0	A	E	D	C	B	... (repeat) ...
CPU 1	B	A	E	D	C	... (repeat) ...
CPU 2	C	B	A	E	D	... (repeat) ...
CPU 3	D	C	B	A	E	... (repeat) ...

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:

CPU 0	A	E	A	A	A	... (repeat) ...
CPU 1	B	B	E	B	B	... (repeat) ...
CPU 2	C	C	C	E	C	... (repeat) ...
CPU 3	D	D	D	D	E	... (repeat) ...

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:

CPU 0	A	A	C	C	A	A	C	C	A	A	C	C	...
CPU 1	B	B	D	D	B	B	D	D	B	B	D	D	...

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:

CPU 0	A	A	A	A	A	A	A	A	A	A	A	A	A	...
CPU 1	B	B	D	D	B	B	D	D	B	B	D	D	...	

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



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

CPU 0														...
CPU 1	B	B	D	D	B	B	D	D	B	B	D	D	...	

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<sup>1</sup>? How do we stop asking questions that are hardly relevant to this otherwise wonderful book?

---

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

### CRUX: HOW TO DEAL WITH LOAD IMBALANCE

How should a multi-queue multiprocessor scheduler handle load imbalance, so as to better achieve its desired scheduling goals?

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

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

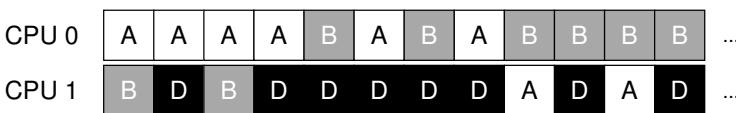


In this case, the desired migration is easy to understand: the OS should simply move one of *B* or *D* to CPU 0. The result of this single job migration is evenly balanced load and everyone is happy.

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



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



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

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

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

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

## 10.6 Linux Multiprocessor Schedulers

Interestingly, in the Linux community, no common solution has approached to building a multiprocessor scheduler. Over time, three different schedulers arose: the O(1) scheduler, the Completely Fair Scheduler (CFS), and the BF Scheduler (BFS)<sup>2</sup>. See Meehan'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.

---

<sup>2</sup>Look up what BF stands for on your own; be forewarned, it is not for the faint of heart.

## References

- [A90] "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors"  
 Thomas E. Anderson  
 IEEE TPDS Volume 1:1, January 1990  
*A classic paper on how different locking alternatives do and don't scale. By Tom Anderson, very well known researcher in both systems and networking. And author of a very fine OS textbook, we must say.*
- [B+10] "An Analysis of Linux Scalability to Many Cores Abstract"  
 Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, Nickolai Zeldovich  
 OSDI '10, Vancouver, Canada, October 2010  
*A terrific modern paper on the difficulties of scaling Linux to many cores.*
- [CSG99] "Parallel Computer Architecture: A Hardware/Software Approach"  
 David E. Culler, Jaswinder Pal Singh, and Anoop Gupta  
 Morgan Kaufmann, 1999  
*A treasure filled with details about parallel machines and algorithms. As Mark Hill humorously observes on the jacket, the book contains more information than most research papers.*
- [FLR98] "The Implementation of the Cilk-5 Multithreaded Language"  
 Matteo Frigo, Charles E. Leiserson, Keith Randall  
 PLDI '98, Montreal, Canada, June 1998  
*Cilk is a lightweight language and runtime for writing parallel programs, and an excellent example of the work-stealing paradigm.*
- [G83] "Using Cache Memory To Reduce Processor-Memory Traffic"  
 James R. Goodman  
 ISCA '83, Stockholm, Sweden, June 1983  
*The pioneering paper on how to use bus snooping, i.e., paying attention to requests you see on the bus, to build a cache coherence protocol. Goodman's research over many years at Wisconsin is full of cleverness, this being but one example.*
- [M11] "Towards Transparent CPU Scheduling"  
 Joseph T. Meehan  
 Doctoral Dissertation at University of Wisconsin—Madison, 2011  
*A dissertation that covers a lot of the details of how modern Linux multiprocessor scheduling works. Pretty awesome! But, as co-advisors of Joe's, we may be a bit biased here.*
- [SHW11] "A Primer on Memory Consistency and Cache Coherence"  
 Daniel J. Sorin, Mark D. Hill, and David A. Wood  
 Synthesis Lectures in Computer Architecture  
 Morgan and Claypool Publishers, May 2011  
*A definitive overview of memory consistency and multiprocessor caching. Required reading for anyone who likes to know way too much about a given topic.*
- [SA96] "Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation"  
 Ion Stoica and Hussein Abdel-Wahab  
 Technical Report TR-95-22, Old Dominion University, 1996  
*A tech report on this cool scheduling idea, from Ion Stoica, now a professor at U.C. Berkeley and world expert in networking, distributed systems, and many other things.*

# Multi-Processor Systems

Much of the discussion in this course has considered the operating system to be running on a time-shared uni-processor ... and this perspective is adequate to fully understand most of those topics. But increasingly many modern computer systems are now multi-processor:

Multiple general purpose CPUs (as opposed to GPUs) that are capable of running unrelated programs or threads (unlike SIMD array processors) and (to some degree) share memory and I/O devices.

These systems are interesting because they are independent enough to encounter many of the problems associated with distributed computing ... but (because they share memory and I/O devices) do things that push the distributed systems envelope. As people develop applications to exploit these platforms, it is important that they understand the issues they present.

## Why Build Multi-Processor Systems

We continue to find applications that require ever more computing power. Sometimes these problems can be solved by horizontally scaled systems (e.g. thousands of web servers). But some problems demand, not more computers, but faster computers. Consider a single huge database, that each year, must handle twice as many operations as it served the previous year. Distributed locking, for so many parallel transactions on a single database, could be prohibitively expensive. The (seemingly also prohibitively expensive) alternative would be to buy a bigger computer every year.

Long ago it was possible to make computers faster by shrinking the gates, speeding up the clock, and improving the cooling. But eventually we reach a point of diminishing returns where physics (the speed of light, information theory, thermodynamics) makes it ever more difficult to build faster CPUs. Recently, most of our improvements in processing speed have come from:

- smarter pipe-lining and increasingly parallel and speculative execution
- putting more cores per chip, more chips per board, and more boards per computer system.

But it is reasonable to ask whether or not 16x3B instructions per second is actually equivalent to 48B instructions per second? The answer (see [Amdahl's Law](#)) depends on whether or not your application can be divided into 16 or more parallelly executable sub-tasks. Fortunately, modern operating systems tend to run large numbers of processes, and expensive computations are increasingly designed to be executable in multiple parallel threads..

For these reasons, multi-processor is the dominant architecture for powerful servers and desktops. And, as the dominant architecture, operating systems must do a good job of exploiting them.

## Multi-Processor Hardware

The above general definition covers a wide range of architectures, that actually have very different characteristics. And so it is useful, to overview the most prominent architectures.

### Hyper-Threading

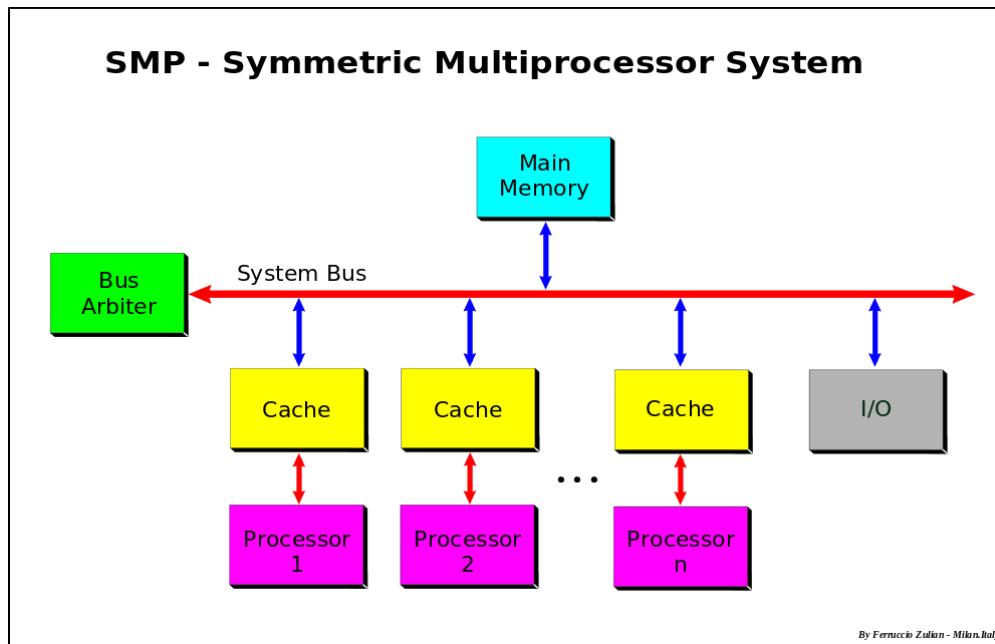
CPUs are much faster than memory. A 2.5GHz CPU might be able to execute more than 5 Billion instructions for second. Unfortunately, 80ns memory can only deliver 12 Million fetches or stores per second. This is almost a 1000x mis-match in performance. The CPU has multiple levels of cache to ensure that we seldom have to go to memory, but even so, the CPU spends a great deal of time waiting for memory.

The idea of hyper-threading is to give each core two sets of general registers, and the ability to run two independent threads. When one of those threads is blocked (waiting for memory) the other thread can be using the execution engine. Think of this as non-preemptive time-sharing at the micro-code level. It is common for a pair of hyper-threads to get 1.2-1.8 times the instructions per second that a single thread would have gotten on the same core. It is theoretically possible to get 2x hyper-threading, but a thread might run out of L1 cache for a long time without blocking, or perhaps both hyper-threads are blocked waiting for memory.

From a performance point-of-view, it is important to understand that both hyper-threads are running in the same core, and so sharing the same L1 and L2 cache. Thus hyper-threads that use the same address space will exhibit better locality, and hence run much better than hyper-threads that use different address spaces.

## Symmetric Multi-Processors

A Symmetric Multi-Processor has some number of cores, all connected to the same memory and I/O busses. Unlike hyper-threads these cores are completely independent execution engines, and (modulo limitations on memory and bus throughput) N cores should be able to execute N times as many instructions per second as a single core.

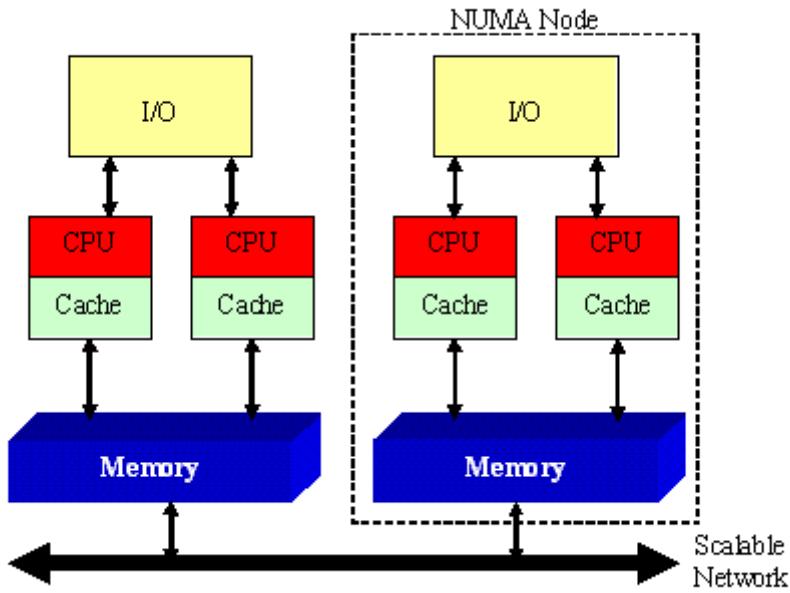


## Cache Coherence

As mentioned previously, much of processor performance is a result of caching. In most SMP systems, each processor has its own L1/L2 caches. This creates a potential *cache-coherency* problem if (for instance) processor 1 updates a memory location whose contents have been cached by processor 2. Program execution based on stale cache entries would result in incorrect results, and so must be prevented. There are a few general approaches to maintaining cache coherency (ensuring that there are no disagreements about the current contents of any cache line), and most SMP systems with per-processor caching incorporate some [Cache Coherency Mechanism](#) to address this issue.

## Cache Coherent Non-Uniform Memory Architectures

It is not feasible to create fast memory controllers that can provide concurrent access to large numbers of cores, and eventually memory bandwidth becomes the bottleneck that prevents scaling to larger numbers of CPUs. A Non-Uniform Memory Architecture addresses this problem by giving each node or CPU its own high-speed local memory, and interconnecting all of the memory busses with a slower but more scalable network.



Operations to local memory may be several times faster than operations to remote memory, and the maximum throughput of the scalable network may be a small fraction of the per-node local memory bandwidth. Such an architecture might provide nearly linear scaling to much larger numbers of processors, but only if we can ensure that most of the memory references are local. The Operating System might be able to deal with the different memory access speeds by trying to allocate memory for each process from the CPU on which that process is running. But there will still be situations where multiple CPUs need to access the same memory. To ensure correct execution, we must maintain coherency between all of the per-node/per-CPU caches. This means that, in addition to servicing remote memory read and write requests, the scalable network that interconnects the nodes must also provide cache coherency. Such architectures are called *Cache Coherent Non-Uniform Memory Architectures* (CC-NUMA), and the implementing networks are called *Scalable Coherent Interconnects*. The best known Scalable Coherent Interconnects are probably Intel's [Quick Path Interconnect](#) (QPI), and AMD's [HyperTransport](#).

## Power Management

The memory and cache interconnections are probably the most interesting part of a multi-processor system, but power management is another very important feature. A multi-core system can consume a huge amount of power ... and most of the time it does not need most of the cores. Many multi-processor systems include mechanisms to slow (or stop) the clocks on unneeded cores, which dramatically reduces system power consumption. This is not a slow process, like a sleep and reboot. A core can be returned to full speed very quickly.

## Multi-Processor Operating Systems

To exploit a multi-processor system, the operating system must be able to concurrently manage multiple threads/processes on each of the available CPU cores. One of the earliest approaches was to run the operating system on one core, and applications on all of the others. This works reasonably for a small number of cores, but as the number of cores increases, the OS becomes the primary throughput bottleneck. Scaling to larger numbers of cores requires the operating system itself to run on multiple cores. Running efficiently on multiple cores requires the operating system to carefully choose which threads/processes to run on which cores and what resources to allocate to them.

When we looked at distributed systems, we saw (e.g. [Deutsch's Seven Falacies](#)) that the mere fact that a network is capable of distributing every operation to an arbitrary node does not make doing so a good idea. It will be seen that the same caveat applies to multi-processor systems.

## Scheduling

If there are threads (or processes) to run, we would like to keep all of the cores busy. If there are not threads (or processes) to run, we would like to put as many cores as possible into low power mode. It is tempting to think that we can just run each thread/process on the next core that becomes available (due to a process blocking or getting a time-slice-end). But some cores may be able to run some threads (or processes) far more efficiently than others.

- dispatching a thread from the same process as the previous thread (to occupy that core) may be much less expensive because re-loading the page table (and flushing all cached TLB entries) is a very expensive operation.
- a thread in the same process may run more efficiently because shared code and data may exploit already existing L1/L2 cache entries.
- threads that are designed to run concurrently (e.g. parallel producer and consumer communicating through shared memory) should be run on distinct cores.

Thus, the choice of when to run which thread in which core is not an arbitrary one. The scheduler must consider what process was last running in each core. It may make more sense to leave one core idle, and delay executing some thread until its preferred core becomes available. The operating system will try to make intelligent decisions, but if the developers understand how work can best be allocated among multi-processor cores, they can advise the operating system with operations like `sched_setaffinity(2)` and `pthread_setaffinity_np(3)`.

## Synchronization

Sharing data between processes is relatively rare in user mode code. But the operating system is full of shared data (process table entries, file descriptors, scheduling and I/O queues, etc). In a uni-processor, the primary causes of race conditions are preemptive scheduling and I/O interrupts. Both of these problems can be managed by disabling (selected) interrupts while executing critical sections ... and many operating systems simply declare that preemptions cannot occur while executing in the operating system.

These techniques cease to be effective once the operating system is running on multiple CPUs in a multi-processor system. Disabling interrupts cannot prevent another core from performing operations on a single global object (e.g. I-node). Thus multi-processor operating systems require some other means to ensure the integrity global data structures. Early multi-processor operating systems tried to create a single, global, kernel lock ... to ensure that only one process at a time could be executing in the operating system. But this is essentially equivalent to running the operating system on a single CPU. As the number of cores increases, the (single threaded) operating system becomes the scalability bottleneck.

The Solution to this problem is finer grained locking. And as the number of cores increased, the granularity required to achieve high parallelism became ever finer. Depending on the particular shared resource and operations, different synchronizations may have to be achieved with different mechanisms (e.g. compare and swap, spin-locks, interrupt disables, try-locks, or blocking mutexes). Moreover for every resource (and combination of resources) we need a plan to prevent deadlock. Changing complex code that involves related updates to numerous data structures to implement fine-grained locking is difficult to do (often requiring significant design changes) and relatively brittle (as maintainers make changes that fail to honor all of the complex synchronization rules). If a decision is made to transition the operating system to finer grained locking, it becomes much more difficult for third party developers to build add-ons (e.g. device drivers and file systems) that will work with the finer grained locking schemes in the hosting operating system.

Because of this complexity, there are relatively few operating systems that are able to efficiently scale to large numbers of multi-processor cores. Most operating systems have chosen simplicity and maintainability over scalability.

## Device I/O

If an I/O operation is to be initiated, does it matter which CPU initiates it? When an I/O interrupt comes in to a multi-processor system, which processor should be interrupted? There are a few reasons we might want to choose carefully which cores handle which I/O operations:

- as with scheduling, sending all operations for a particular device to a particular core may result in more L1/L2 cache hits and more efficient execution.
- synchronization between the synchronous (resulting from system calls) and asynchronous (resulting from interrupts) portions of a device driver if they are all executing in the same CPU.
- each CPU has a limited I/O throughput, and we may want to balance activity among the available cores.
- some CPUs may be bus-wise closer to some I/O devices, so that operations go more quickly initiated from some cores.

Many multi-processor architectures have interrupt controllers that are configurable for which interrupts should be delivered to which processors.

## Non-Uniform Memory Architectures

CC-NUMA is only viable if we can ensure that the vast majority of all memory references can be satisfied from local memory. Doing this turns out to create a lot of complexity for the operating system.

When we were discussing uni-processor memory allocation, we observed that significant savings could be achieved if we shared a single copy of a (read only) load module among all processes that were running that program. This ceases to be true when those processes are running on distinct NUMA nodes. Code and other read-only data should have a separate copy (in local memory) on each NUMA node. The cost (in terms of wasted memory) is negligible in comparison performance gains from making all code references local.

When a program calls *fork(2)* to create a new process, *exec(2)* to run a new program, or *sbrk(2)* to expand its address space, the required memory should always be allocated from the node-local memory pool. This creates a very strong affinity between processes and NUMA nodes. If it is necessary to migrate a process to a new NUMA node, all of its allocated code and data segments should be copied into local memory on the target node.

As noted above, the operating system is full of data structures that are shared among many cores. How can we reduce the number or cost of remote memory references associated with those shared data structures? If the updates are few, sparse and random, there may be little we can do to optimize them ... but their costs will not be high. When more intensive use of shared data structures is required, there are two general approaches:

1. move the data to the computation
  - lock the data structure.
  - copy it into local memory.
  - update the global pointer to reflect its new location.
  - free the old (remote) copy.
  - perform all subsequent operations on the (now) local copy.
2. move the computation to the data
  - look up the node that owns the resource in question.
  - send a message requesting it to perform the required operations.
  - await a response.

In practice, both of these techniques are used ... the choice determined by the particulars of the resource and its access.

As with fine-grained synchronization of kernel data structures, this turns out to be extremely complicated. Relatively few operating systems have been willing to pay this cost, and so (again) most opt for simplicity and maintainability over performance and scalability.

# Eventual Consistency

Wednesday, November 30, 2016 4:32 PM

Eventual consistency

Eventual consistency is a [consistency model](#) used in [distributed computing](#) to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.<sup>[1]</sup> Eventual consistency is widely deployed in distributed systems, often under the moniker of [optimistic replication](#),<sup>[2]</sup> and has origins in early mobile computing projects.<sup>[3]</sup> A system that has achieved eventual consistency is often said to have **converged**, or achieved **replica convergence**.<sup>[4]</sup> Eventual consistency is a weak guarantee – most stronger models, like [linearizability](#) are trivially eventually consistent, but a system that is merely eventually consistent does not usually fulfill these stronger constraints.

Eventually consistent services are often classified as providing BASE (Basically Available, Soft state, Eventual consistency) semantics, in contrast to traditional [ACID](#)

([Atomicity](#), [Consistency](#), [Isolation](#), [Durability](#)) guarantees.<sup>[5][6]</sup>

Eventual consistency is sometimes criticized<sup>[7]</sup> as increasing the complexity of distributed software applications. This is partly because eventual consistency is purely a [liveness](#) guarantee (reads eventually return the same value) and does not make [safety](#) guarantees: an eventually consistent system can return any value before it converges.

Conflict resolution

In order to ensure replica convergence, a system must reconcile differences between multiple copies of distributed data. This consists of two parts:

- exchanging versions or updates of data between servers (often known as [anti-entropy](#)),<sup>[8]</sup> and
- choosing an appropriate final state when concurrent updates have occurred, called [reconciliation](#).

The most appropriate approach to reconciliation depends on the application. A widespread approach is "*last writer wins*".<sup>[1]</sup> Another is to invoke a user-specified conflict handler.<sup>[4]</sup> [Timestamps](#) and [vector](#)

[CLOCKS](#) are often used to detect concurrency between updates.

Reconciliation of concurrent writes must occur sometime before the next read, and can be scheduled at different instants:[\[3\]](#)[\[9\]](#)

- Read repair: The correction is done when a read finds an inconsistency. This slows down the read operation.
- Write repair: The correction takes place during a write operation, if an inconsistency has been found, slowing down the write operation.
- Asynchronous repair: The correction is not part of a read or write operation.

Strong eventual consistency

Whereas eventual consistency is only a [liveness](#) guarantee (updates will be observed eventually), **strong eventual consistency** (SEC) adds the [safety](#) guarantee that any two nodes that have received the same (unordered) set of updates will be in the same state. If, furthermore, the system is [monotonic](#), the application will never suffer rollbacks. [Conflict-free replicated data types](#) are a common approach to ensuring SEC.[\[10\]](#)

## Cluster

There are many different types of clusters. Common types include:

- load sharing clusters, which divide work among the members.
- high availability clusters, where back-up nodes take over when primary nodes fail.
- information sharing clusters, which ensure the dissemination of information throughout a network.

There are so many different goals and architectures that Greg Phister (in "In Search of Clusters") came to the conclusion that it is very difficult to even define the term. About the only thing we can say for certain is that a cluster is a networked connection of nodes, all of whom agree that they are part of a cluster.

## Membership

If a cluster is defined as a networked connection of nodes who consider themselves to be participants in the cluster, then obviously "membership" is a key concept.

We can distinguish two types of membership:

- potential, eligible or designated members
- active or currently participating members

This distinction is important because only active members can communicate with one-another. Thus it is that the term "membership" is most commonly used to describe only the currently participating members.

It is very important to know who the current (active) cluster members are. We may, for instance, be required to make sure that each of them has been informed of some operation before we are allowed to perform it. Cluster membership often comes with responsibilities (e.g. a guarantee to respond to certain requests within a certain period if time). Thus it is vital that we know when nodes enter and leave the cluster.

In most clusters, a node has to be explicitly configured or provisioned into the cluster ... so that the set of potential members is well known, and perhaps even closed to new members. There are, however, some types of clusters where any node is welcome to join at any time.

## Node Redundancy

In a clustered system, work is divided among the active members. To reduce distributed synchronization, it is common to partition the work (e.g. designate each server responsible for a certain subset (e.g. a file system, a range of keys, etc) of requests, and route all requests to their designated owner). In such systems, we can talk about *primaries* (the designated owners) and *secondaries* (nodes who are prepared to take over for a primary if he fails).

There are three fundamentally different approaches to take to high availability:

- Active/Stand-By

The system is divided into *active* and *stand-by* nodes. The incoming requests are partitioned among the active nodes. The stand-by nodes are idle until an active node fails, at which point a stand-by node takes over his work.

- Active/Active

The incoming requests are partitioned among all of the available nodes. If one node fails, his work will be redistributed among the survivors.

An active/active architecture achieves better resource utilization, and so may be more cost-effective. But when a failure occurs, the load on the surviving nodes is increased and so performance may suffer. An active/stand-by architecture normally has idle capacity, but may not suffer any performance degradation after a failure.

We can also look at how quickly a successor is able to take over a failed node's responsibilities. In some architectures, all operations are mirrored to the secondaries, enabling them to very quickly assume the primary role. Such secondaries are called *hot standbys*. In other systems, the secondary waits to be notified of the primary's failure, after which it opens and reads the failed primary's last check-point or journal. The former approach results in more network and CPU load, but much faster fail-overs. The latter approach consumes fewer resources during normal operation, but may take much longer to resume service after a primary has failed.

## Heart Beat

Ideally nodes will announce the fact that they are joining the cluster, or are about to leave it. This is not always the case:

1. a system may crash.
2. the clustering applications may crash.
3. a node may become so busy that the clustering applications cannot run.
4. a network interface or link may fail.

Since we cannot be sure that member will notify the other members before he leaves the cluster, we need a means of detecting a member who has dropped unexpectedly out of the cluster. The most common technique is called a "heart beat". A heart beat is a message, regularly exchanged between cluster members. These may be special messages, or they may be piggy-backed on other traffic. If too much time passes without our having received a heart-beat from some node, we will assume that node has failed, and is no longer an active cluster member.

The failure of a node may (in some clusters) have serious consequences (e.g. the freeing of all resources allocated to that node, and the aborting of all in progress transactions from that node). To prevent "false alarms", many systems perform heart-beats over multiple channels, or have a back-up link with which they attempt to confirm a failure before reporting a node to be dead.

## Cluster Master and Election

It is often convenient to elect or designate one node to be the cluster master:

- Coming to a mutual agreement between multiple nodes can be a complex process (e.g. Three Phase Commits). If one node is designated a cluster master, that node can serve as a central point of synchronization and/or control for operations in the cluster.
- Rather than requiring all nodes to heart-beat one-another, it is more economical to simply have all nodes heart-beat with the cluster master. He will detect the failure of any other node, and all nodes will detect his failure.

The election of a cluster master may, itself, be a complex process ... but having performed that process may eliminate the need for any further negotiations. There are numerous well established election/concensus algorithms. One of the best known is Leslie Lamport's [Paxos algorithm](#).

## Split Brain

A pathological network failure might divide a cluster into multiple sub-clusters, which cannot communicate with one-another. Such an event is sometimes referred to as a "partitioning" of the network. If the cluster manages critical resources (e.g. a database or nuclear warhead), it is possible that the independent sub-clusters will all continue operating and make independent but incompatible decisions. Such a condition is called "split-brain" (as if two halves of our brain were working independently and at cross-purposes).

There are two standard approaches to preventing "split-brain":

- a. quorum
- b. voting devices

## Quorum

If there are  $N$  potential members in a cluster, we can build in a rule that says a cluster cannot form with fewer than  $(N/2)+1$  members. This accomplishes two purposes:

- It makes it impossible for any partitioning to result in two viable sub-clusters (because  $N$  nodes cannot be divided into two groups that both contain at least  $(N/2)+1$  nodes).
- It ensures that any decision made (and persisted) by this quorum will be remembered by any future quorum (because any group of  $(N/2)+1$  nodes will have at least one member in common with every other group of  $(N/2)+1$  nodes that has ever existed).

The problem with using a numerical quorum is that if  $(N/2)+1$  nodes have been damaged, it will be impossible for the surviving nodes to form a new cluster ... even if there is no split-brain.

## Voting Devices

If there is a single piece of hardware in the cluster, that must be present for the cluster to function, and that can only be owned by one node, that device can be used as a voting device.

Consider, for instance, a shared disk. If that disk is absolutely required to provide service, a node that no longer has access to that disk cannot provide service (and hence is not eligible to form a cluster). But what if two nodes can both talk to the disk, but cannot communicate with one-another? They may be able to use the disk as a voting device ... e.g. by writing a recent time-stamp into a well-known block.

Some clusters include resources that can easily serve as voting devices. There are also specially built (very reliable) voting devices that exist solely for this purpose. If there is a voting device, a cluster could be formed by a single node ... because the voting device would prevent split-brain.

## Fencing

What if, you were not only suffering from schizophrenia, but the other side of your brain had actually gone rogue, and was trying to commit acts of mayhem against you and others? In some clustered systems, it must be assumed that if a node has fallen out of the cluster, he is no longer trust-worthy ... and must be "fenced-out" of the cluster. There are two common approaches to fencing:

- reservable devices  
Some devices can be told which interface to listen to, and not to listen to the other interface. This is often done with dual-ported disks. The node that has seized the quorum device will then instruct the quorum device not to accept commands from any other node.
- remote power control  
Some clustered systems come with remote power controllers, and a node that has seized control of the

cluster from a previous (apparently failed) cluster master will often power-off or reset the previous master, to ensure that he does not continue to vie for control of the cluster and its resources.

# Horizontally Scalable Systems

The term *horizontally scalable* refers to systems whose capacity and throughput are increased by adding additional nodes. This is in distinction to *vertically scaled* systems, where adding capacity and throughput generally involves replacing smaller nodes with larger and more powerful ones.

It seems that most technological advancements involve building increasingly complex systems that adapt to increasingly subtle inputs. But we should also recognize that improved understandings and technologies often enable us to build systems that are simultaneously superior to, and in many ways, much simpler than their predecessors. Horizontally scaled systems are based on protocols developed for client-server distributed storage and distributed computing systems. But, unlike their more complex cousins, their evolution has guided by the principles of maximum independence (loose coupling) and parallelism.

Not all problems can be solved with a horizontally scalable architecture. But where they work, they work very well. This makes horizontally scaled architectures well-worthy of study.

## Goals of Horizontally Scalable Systems

These systems started with many of the same goals shared by most distributed systems: scalability of capacity and throughput, reliability, and availability. But whereas earlier distributed systems attempted to provide single-system services and semantics, horizontally scaled systems offer simplified (and decidedly non-Posix) semantics. Most of these simplifications have a few basic (and highly inter-related) goals:

- improve scalability and robustness by eliminating the need for synchronization and communication between parallel servers.
- improve flexibility and performance by enabling the (non-disruptive) addition or removal of servers at any time.
- stateless protocols that permit requests to be arbitrarily distributed and redistributed among those servers.
- turn servers into standardized components, easily deployed and requiring little or no configuration.
- service protocols that are designed to minimize the potential modes of failure and enable simple recoveries.
- service protocols that are optimized for throughput (vs latency) over very large (e.g. continental) distances.
- service protocols that exploit the numerous optimizations and other features that have come to be standard in networks that serve heavy HTTP traffic.

And, as a (not unintentional) side-effect, these systems are much simpler to design, maintain, deploy, and manage than other (more complex) distributed systems.

## Horizontally Scalable Hardware Platforms

All services in horizontally scalable systems are exchanged via network messages. A node in such a system is simply a computer that implements the specified protocols. The instruction set, operating system, memory capacity, and storage technologies are merely implementation details ... of little importance to the system design. In fact, ignoring deployment issues, there should be no problem running a service where every one of the server nodes was a different instruction set, running a different operating system/version, and using different network interfaces and storage controllers.

This high degree of platform independence has made it very common to run horizontally scalable systems on virtual machines. When a new server is needed, we simply clone a standard image, and boot up a new virtual machine. Thirty seconds later, a new server is running. A service run on a virtual machine may be a little less

efficient than it would be if run on physical hardware, but the flexibility and ease of deployment and capacity balancing greatly outweigh any reduced efficiency.

Note, however, that it is not the case that all virtual servers are equivalent to generic desktops. High end VM servers may include GPUs, NVRAM, flash memory, or other special hardware than can be allocated to VMs. With access to such acceleration hardware, a service running in a virtual machine may be able to rival the performance of (much more expensive) custom-built hardware.

If a horizontally scalable system is comprised of many servers, and new servers can be added at any time, it is essential that they all have appropriate network connectivity (for client-facing traffic, back-side servers, and peer-to-peer replication). It is not practical to rack new switches and connect new cables each time a new server is to be integrated into a particular service. Rather, large computing facilities are moving towards *Software Defined Networking* where all servers (virtual or real) are connected to large and versatile switches that can easily be reconfigured to create whatever virtual network topologies and capacities are required for the service to be provided.

Storage is a similar problem. New virtual machines need virtual disks for the operating system to boot off of, and to store the content they will manage. Large computing facilities are also moving towards *Software Defined Storage*, where intelligent storage controllers draw on pools of physical disks (or SSDs) to create virtual disks (LUNs) with the required capacity, redundancy, and performance. The new servers have no idea what the underlying storage is, but merely use remote disk access protocols (e.g. Fibre Channel or iSCSI) to access configured storage.

## Horizontally Scalable System Architectures

The individual servers in a horizontally scaled architecture are highly independent. They do not share CPUs, disks, network interfaces, or a single operating system instance. This means it is unlikely that any single failure would affect multiple servers. Even enclosure components (e.g. fans and power supplies) or networking components (e.g. switches and routers) that might be shared by multiple servers have redundancy. Such systems are said to have no *single point of failure* or a *shared nothing architecture*. Both mean that there is no single (shared) component whose failure could affect multiple systems. Such design is key to reliability and availability.

Even if there are no single points of failure in the computing and storage components, all of the servers in a single room (or campus) could potentially be taken down by a regional disaster (power failure, flood, earthquake, etc). For this reason, highly available, and highly reliable systems often make copies and include back-up servers that are far away (e.g. in other cities). Far separated facilities that are unlikely to be affected by a single regional disaster are often referred to as being in distinct *availability zones*. If the servers in one availability zone go down, the work can be handed off to servers in a different availability zone. The ability to fail over to distant copies and serve is sometimes called *geographic disaster recovery*.

## Horizontally Scalable Software Architectures

The classic horizontally scaled system is a farm of web-servers, all of which have access to/copies of the same content. As demand increases, more servers are brought on-line. When each server starts up, it is made known to a network switch that starts sending its requests.

- Since HTTP is a stateless protocol, the switch can route any request to any server. This routing may be as simple as round-robin, or it may be load-balanced based on measured response times to recent requests.
- Parallel servers have no need to communicate with one-another, and the only configuration a web server requires is knowing where to find the content it is serving.
- If a web-server crashes, the switch will stop sending it new requests. If the operations in the service protocol are idempotent, the client will time-out and retransmit the request, which will automatically be

routed to a different server.

Horizontally scalable systems do not have to be stateless. RESTful interfaces enable stateful interactions with all of the the same advantages. Even non-RESTful services can benefit from horizontal scaling. Consider the *shopping carts* supported by many product web sites. The front-end web-servers are indeed horizontally scaled and stateless. Shopping cart display and update operations are forwarded from the responding web-server to a back-end data-base server. The back-end data-base server may be highly stateful, and not at all horizontally scalable. But if 99.9% of all requests can be satisfied by the front line of web servers, it may be much easier to build a (perhaps vertically scaled) data-base server to handle the shopping cart requests.

Even (highly stateful) storage systems (including databases) can be designed to be horizontally scalable. Distributed Key/Value Stores often divide the key namespace into a independent subsets (usually called *shards*), each of which is assigned to a small group (e.g. 3) of servers. Assigning multiple servers to each shard enables us to maintain data availability even if some servers fail. Imposing a low cap on the number of servers involved in any particular transaction enables the protocols to scale to thousands of nodes with no degradataion in performance. When we need to add capacity, we do not add more servers for a particular shard; Rather we increase the number of shards into which the name space is divided, and assign the new shards to new servers. Similar techniques have been employed to improve the scalability of block storage, object storage, and even file storage services.

## Cloud Model

The *cloud model* may be more a business model (renting services from an independent service provider, vs. buying equipment) than an architecture, but it is very well aligned with horizontally scalable architectures:

- All services are delivered via (well standardized) network protocols.
- The cloud model opaquely encapsulates the individual servers behind a single highly available IP address. This enables the network to distribute requests among the available servers.
- The resources presented by a cloud service are intended to be abstract/logical and thin-provisioned. Horizontally scaled architectures are well suited to implementing and delivering such services.
- The flexibility to continuously add, remove, and rebalance resources in a horizontally scalable system, makes it possible for a cloud service provider to easily accommodate great fluctuations in demand.
- Public cloud services are (in most cases) unlikely to be co-located with the clients, and so the service protocols must be optimized for (throughput) efficiency over WAN-scale communications links.
- Horizontally scalable systems' ease of deployment/management yields economies of scale that make it possible for cloud service providers to offer their services at very competitive prices. Consider, for instance, the cost of maintaining redundant computing facilities in many different cities. This is normal operation for a large service provider, but would be prohibitively expensive for the typical company to do for itself.

But it is worth noting that the cloud model does not necessarily mean that all data access becomes remote. We always have a choice:

- a. send the data to the computation (e.g. remote data access)
- b. send the computation to the data (e.g. Map/Reduce)

Many cloud storage providers also also offer Virtual Machine hosting, and for applications that run in those remote virtual machines, all cloud data access will be local. For many applications, the amount of raw data processed is much greater than the amount of output that is produced. In such cases, it may be both faster and more economical to run the computation on a VM server that is co-located with the data it will process.