

Lecture 1: Introduction to Operating Systems

What is an operating system?

- Middleware between user programs and system hardware
- Manages hardware: CPU, main memory, IO devices (disk, network card, mouse, keyboard etc.)

What happens when you run a program? (Background)

- A compiler translates high level programs into an executable (“.c” to “a.out”)
- The exe contains instructions that the CPU can understand, and data of the program (all numbered with addresses)
- Instructions run on CPU: hardware implements an instruction set architecture (ISA)
- CPU also consists of a few registers, e.g.,
 - Pointer to current instruction (program counter or PC)
 - Operands of instructions, memory addresses

So, what happens when you run a program?

- To run an exe, CPU
 - fetches instruction pointed at by PC from memory
 - loads data required by the instructions into registers
 - decodes and executes the instruction
 - stores results to memory
- Most recently used instructions and data are in CPU caches for faster access

So, what does the OS do?

- OS manages program memory
 - Loads program executable (code, data) from disk to memory
- OS manages CPU
 - Initializes program counter (PC) and other registers to begin execution
- OS manages external devices
 - Read/write files from disk.

OS manages CPU

- OS provides the process abstraction
 - Process: a running program
 - OS creates and manages processes
- Each process has the illusion of having the complete CPU, i.e., OS virtualizes CPU
- Timeshares CPU between processes
- Enables coordination between processes

OS manages memory

- OS manages the memory of the process: code, data, stack, heap etc
- Each process thinks it has a dedicated memory space for itself, numbers code and data starting from 0 (virtual addresses)
- OS abstracts out the details of the actual placement in memory, translates from virtual addresses to actual physical addresses

OS manages devices

- OS has code to manage disk, network card, and other external devices: device drivers
- Device driver talks the language of the hardware devices
 - Issues instructions to devices (fetch data from a file)
 - Responds to interrupt events from devices (user has pressed a key on keyboard)
- Persistent data organized as a filesystem on disk

Design goals of an operating system

- Convenience, abstraction of hardware resources for user programs
- Efficiency of usage of CPU, memory, etc.
- Isolation between multiple processes

History of operating systems

- Started out as a library to provide common functionality across programs
- Later, evolved from procedure call to system call: what's the difference?
- When a system call is made to run OS code, the CPU executes at a higher privilege level
- Evolved from running a single program to multiple processes concurrently

The Process Abstraction

OS provides process abstraction

- When you run an exe file, the OS creates a process = a running program
- OS timeshares CPU across multiple processes: virtualizes CPU
- OS has a CPU scheduler that picks one of the many active processes to execute on a CPU
 - Policy: which process to run
 - Mechanism: how to “context switch” between processes

What constitutes a process?

- A unique identifier (PID)
- Memory image
 - Code & data (static)
 - Stack and heap (dynamic)
- CPU context: registers
 - Program counter
 - Current operands
 - Stack pointer
- File descriptors
 - Pointers to open files and devices

How does OS create a process?

- Allocates memory and creates memory image
 - Loads code, data from disk exe
 - Creates runtime stack, heap
- Opens basic files
 - STD IN, OUT, ERR
- Initializes CPU registers
 - PC points to first instruction

States of a process

- Running: currently executing on CPU
- Ready: waiting to be scheduled
- Blocked: suspended, not ready to run
 - Why? Waiting for some event, e.g., process issues a read from disk
 - When is it unblocked? Disk issues an interrupt when data is ready
- New: being created, yet to run
- Dead: terminated

Process State Transitions

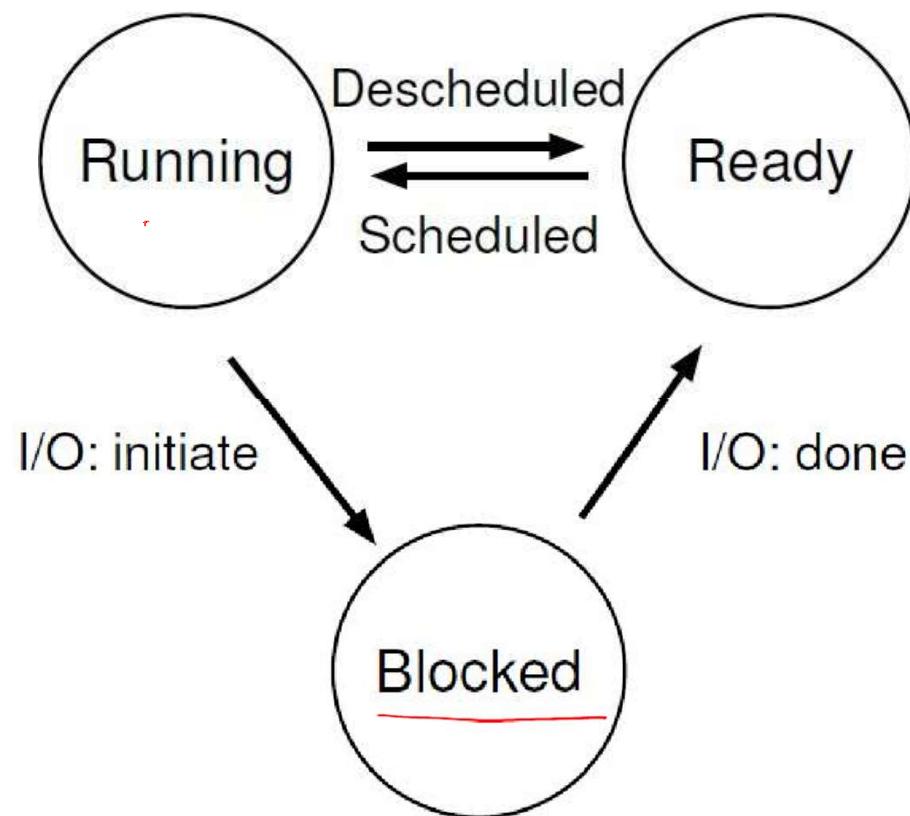


Figure 4.2: Process: State Transitions

Example: Process States

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

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

OS data structures

- OS maintains a data structure (e.g., list) of all active processes
- Information about each process is stored in a process control block (PCB)
 - Process identifier
 - Process state
 - Pointers to other related processes (parent)
 - CPU context of the process (saved when the process is suspended)
 - Pointers to memory locations
 - Pointers to open files

The Abstraction: The Process

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

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

THE CRUX OF THE PROBLEM:

HOW TO PROVIDE THE ILLUSION OF MANY CPUs?

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

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

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

TIP: USE TIME SHARING (AND SPACE SHARING)

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

switch, which gives the OS the ability to stop running one program and start running another on a given CPU; this **time-sharing** mechanism is employed by all modern OSes.

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

4.1 The Abstraction: A Process

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

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

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

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

Note that there are some particularly special registers that form part of this machine state. For example, the **program counter** (PC) (sometimes called the **instruction pointer** or IP) tells us which instruction of the program will execute next; similarly a **stack pointer** and associated **frame**

TIP: SEPARATE POLICY AND MECHANISM

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

pointer are used to manage the stack for function parameters, local variables, and return addresses.

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

4.2 Process API

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

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

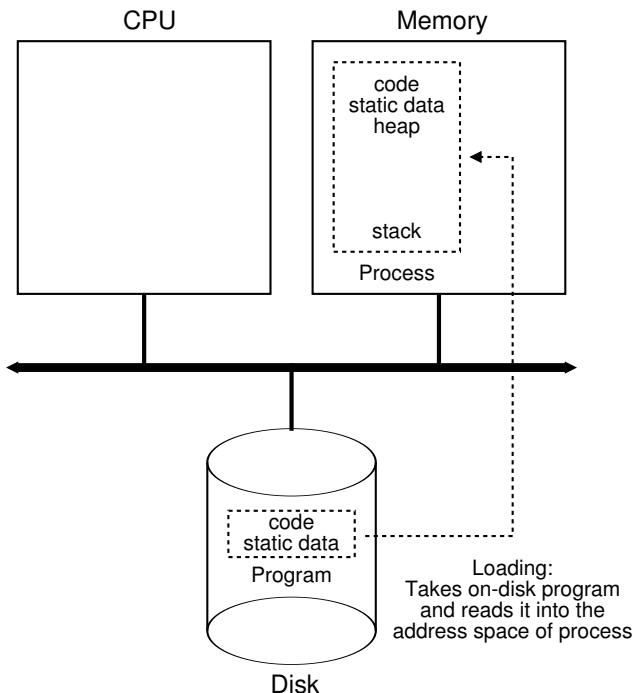


Figure 4.1: Loading: From Program To Process

4.3 Process Creation: A Little More Detail

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

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

In early (or simple) operating systems, the loading process is done **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 and print output to the screen. We'll learn more about I/O, file descriptors, and the like in the third part of the book on **persistence**.

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

4.4 Process States

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

- **Running:** In the running state, a process is running on a processor. This means it is executing instructions.
- **Ready:** In the ready state, a process is ready to run but for some reason the OS has chosen not to run it at this given moment.

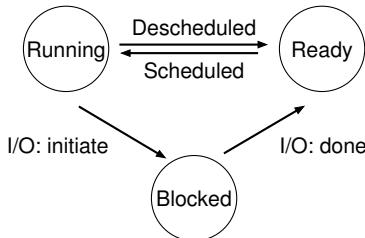


Figure 4.2: Process: State Transitions

- **Blocked:** In the blocked state, a process has performed some kind of operation that makes it not ready to run until some other event takes place. A common example: when a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

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

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

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

Figure 4.3: Tracing Process State: CPU Only

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

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

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, Process₀ initiates an I/O and becomes blocked waiting for it to complete; processes become blocked, for example, when reading from a disk or waiting for a packet from a network. The OS recognizes Process₀ is not using the CPU and starts running Process₁. While Process₁ is running, the I/O completes, moving Process₀ back to ready. Finally, Process₁ finishes, and Process₀ 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 Process₁ while Process₀ issued an I/O; doing so improves resource utilization by keeping the CPU busy. Second, the system decided not to switch back to Process₀ when its I/O completed; it is not clear if this is a good decision or not. What do you think? These types of decisions are made by the **OS scheduler**, a topic we will discuss a few chapters in the future.

4.5 Data Structures

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

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

From the figure, you can see a couple of important pieces of information the OS tracks about a process. The **register context** will hold, for a

```

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

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

// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;                                // Start of process memory
    uint sz;                                   // Size of process memory
    char *kstack;                             // Bottom of kernel stack
                                              // for this process
    enum proc_state state;                    // Process state
    int pid;                                  // Process ID
    struct proc *parent;                     // Parent process
    void *chan;                               // If !zero, sleeping on chan
    int killed;                              // If !zero, has been killed
    struct file *ofile[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

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

You can also see from the figure that there are some other states a process can be in, beyond running, ready, and blocked. Sometimes a system will have an **initial** state that the process is in when it is being created. Also, a process could be placed in a **final** state where it has exited but

ASIDE: DATA STRUCTURE — THE PROCESS LIST

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

has not yet been cleaned up (in UNIX-based systems, this is called the **zombie state**¹). This final state can be useful as it allows other processes (usually the **parent** that created the process) to examine the return code of the process and see if the just-finished process executed successfully (usually, programs return zero in UNIX-based systems when they have accomplished a task successfully, and non-zero otherwise). When finished, the parent will make one final call (e.g., `wait()`) to wait for the completion of the child, and to also indicate to the OS that it can clean up any relevant data structures that referred to the now-extinct process.

4.6 Summary

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

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

ASIDE: KEY PROCESS TERMS

- The **process** is the major OS abstraction of a running program. At any point in time, the process can be described by its state: the contents of memory in its **address space**, the contents of CPU registers (including the **program counter** and **stack pointer**, among others), and information about I/O (such as open files which can be read or written).
- The **process API** consists of calls programs can make related to processes. Typically, this includes creation, destruction, and other useful calls.
- Processes exist in one of many different **process states**, including running, ready to run, and blocked. Different events (e.g., getting scheduled or descheduled, or waiting for an I/O to complete) transition a process from one of these states to the other.
- A **process list** contains information about all processes in the system. Each entry is found in what is sometimes called a **process control block (PCB)**, which is really just a structure that contains information about a specific process.

References

- [BH70] "The Nucleus of a Multiprogramming System" by Per Brinch Hansen. Communications of the ACM, Volume 13:4, April 1970. *This paper introduces one of the first microkernels in operating systems history, called Nucleus. The idea of smaller, more minimal systems is a theme that rears its head repeatedly in OS history; it all began with Brinch Hansen's work described herein.*
- [CK+08] "The xv6 Operating System" by Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich. From: <https://github.com/mit-pdos/xv6-public>. *The coolest real and little OS in the world. Download and play with it to learn more about the details of how operating systems actually work. We have been using an older version (2012-01-30-1-g1c41342) and hence some examples in the book may not match the latest in the source.*
- [DV66] "Programming Semantics for Multiprogrammed Computations" by Jack B. Dennis, Earl C. Van Horn. Communications of the ACM, Volume 9, Number 3, March 1966 . *This paper defined many of the early terms and concepts around building multiprogrammed systems.*
- [L+75] "Policy/mechanism separation in Hydra" by R. Levin, E. Cohen, W. Corwin, F. Pollack, W. Wulf. SOSP '75, Austin, Texas, November 1975. *An early paper about how to structure operating systems in a research OS known as Hydra. While Hydra never became a mainstream OS, some of its ideas influenced OS designers.*
- [V+65] "Structure of the Multics Supervisor" by V.A. Vyssotsky, F. J. Corbato, R. M. Graham. Fall Joint Computer Conference, 1965. *An early paper on Multics, which described many of the basic ideas and terms that we find in modern systems. Some of the vision behind computing as a utility are finally being realized in modern cloud systems.*

Homework (Simulation)

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

Questions

1. Run `process-run.py` with the following flags: `-l 5:100,5:100`. What should the CPU utilization be (e.g., the percent of time the CPU is in use?) Why do you know this? Use the `-c` and `-p` flags to see if you were right.
2. Now run with these flags: `./process-run.py -l 4:100,1:0`. These flags specify one process with 4 instructions (all to use the CPU), and one that simply issues an I/O and waits for it to be done. How long does it take to complete both processes? Use `-c` and `-p` to find out if you were right.
3. Switch the order of the processes: `-l 1:0,4:100`. What happens now? Does switching the order matter? Why? (As always, use `-c` and `-p` to see if you were right)
4. We'll now explore some of the other flags. One important flag is `-S`, which determines how the system reacts when a process issues an I/O. With the flag set to `SWITCH.ON-END`, the system will NOT switch to another process while one is doing I/O, instead waiting until the process is completely finished. What happens when you run the following two processes (`-l 1:0,4:100 -c -S SWITCH.ON-END`), one doing I/O and the other doing CPU work?
5. Now, run the same processes, but with the switching behavior set to switch to another process whenever one is WAITING for I/O (`-l 1:0,4:100 -c -S SWITCH.ON.IO`). What happens now? Use `-c` and `-p` to confirm that you are right.
6. One other important behavior is what to do when an I/O completes. With `-I IO.RUN.LATER`, when an I/O completes, the process that issued it is not necessarily run right away; rather, whatever was running at the time keeps running. What happens when you run this combination of processes? (Run `./process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH.ON.IO -I IO.RUN.LATER -c -p`) Are system resources being effectively utilized?
7. Now run the same processes, but with `-I IO.RUN.IMMEDIATE` set, which immediately runs the process that issued the I/O. How does this behavior differ? Why might running a process that just completed an I/O again be a good idea?

8. Now run with some randomly generated processes: `-s 1 -1 3:50,3:50` or `-s 2 -1 3:50,3:50` or `-s 3 -1 3:50,3:50`. See if you can predict how the trace will turn out. What happens when you use the flag `-I IO_RUN_IMMEDIATE` vs. `-I IO_RUN_LATER`? What happens when you use `-S SWITCH_ON_IO` vs. `-S SWITCH_ON_END`?

Process API

What API does the OS provide to user programs?

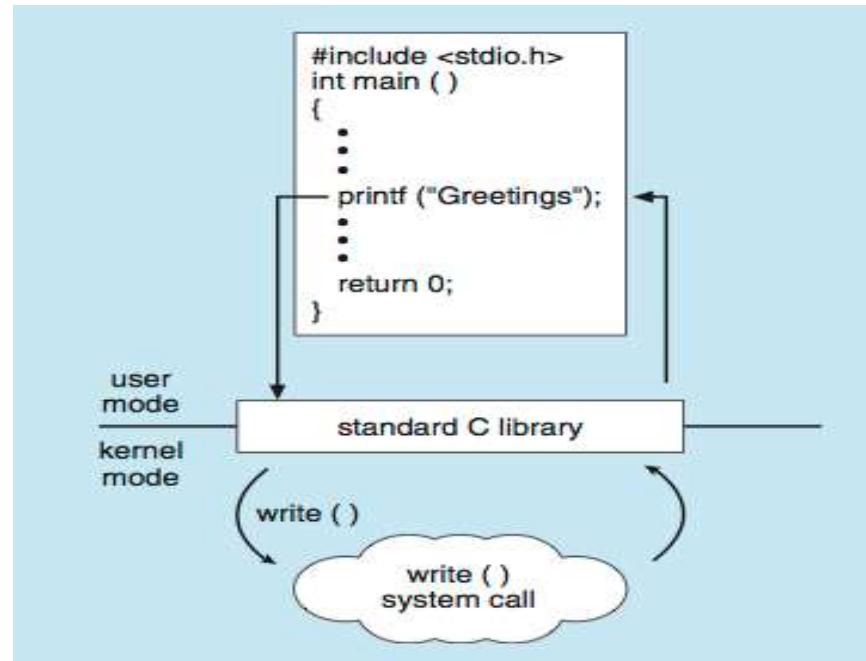
- API = Application Programming Interface
= functions available to write user programs
- API provided by OS is a set of “system calls”
 - System call is a function call into OS code that runs at a higher privilege level of the CPU
 - Sensitive operations (e.g., access to hardware) are allowed only at a higher privilege level

So, should we rewrite programs for each OS?

- POSIX API: a standard set of system calls that an OS must implement
 - Programs written to the POSIX API can run on any POSIX compliant OS
 - Most modern OSes are POSIX compliant
 - Ensures program portability
- Program language libraries hide the details of invoking system calls
 - The `printf` function in the C library calls the `write` system call to write to screen
 - User programs usually do not need to worry about invoking system calls

Example -- Standard C Library

C program invoking printf() library call, which calls write() system call



Process related system calls (in Unix)

- `fork()` creates a new child process
 - All processes are created by forking from a parent
 - The `init` process is ancestor of all processes
- `exec()` makes a process execute a given executable
- `exit()` terminates a process
- `wait()` causes a parent to block until child terminates
- Many variants exist of the above system calls with different arguments

What happens during a fork?

- A new process is created by making a copy of parent's memory image
- The new process is added to the OS process list and scheduled
- Parent and child start execution just after fork (with different return values)
- Parent and child execute and modify the memory data independently

```
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>
```

Waiting for children to die...

- Process termination scenarios
 - By calling `exit()` (`exit` is called automatically when end of main is reached)
 - OS terminates a misbehaving process
- Terminated process exists as a zombie
- When a parent calls `wait()`, zombie child is cleaned up or “reaped”
- `wait()` blocks in parent until child terminates (non-blocking ways to invoke `wait` exist)
- What if parent terminates before child? init process adopts orphans and reaps them

```
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**)

What happens during exec?

- After fork, parent and child are running same code
 - Not too useful!
- A process can run `exec()` to load another executable to its memory image
 - So, a child can run a different program from parent
- Variants of `exec()`, e.g., to pass commandline arguments to new executable

```
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)

Case study: How does a shell work?

- In a basic OS, the `init` process is created after initialization of hardware
- The `init` process spawns a shell like `bash`
- Shell reads user command, forks a child, execs the command executable, waits for it to finish, and reads next command
- Common commands like `ls` are all executables that are simply exec'ed by the shell

```
prompt>ls  
a.txt b.txt c.txt
```

More funky things about the shell

- Shell can manipulate the child in strange ways
- Suppose you want to redirect output from a command to a file
- prompt>ls > foo.txt
- Shell spawns a child, rewrites its standard output to a file, then calls exec on the child

```

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)

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

```

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

```

Interlude: Process API

ASIDE: INTERLUDES

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

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

CRUX: HOW TO CREATE AND CONTROL PROCESSES

What interfaces should the OS present for process creation and control? How should these interfaces be designed to enable powerful functionality, ease of use, and high performance?

5.1 The `fork()` System Call

The `fork()` system call is used to create a new process [C63]. However, be forewarned: it is certainly the strangest routine you will ever call¹. 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!

¹Well, 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 main(int argc, char *argv[]) {
6     printf("hello world (pid:%d)\n", (int) getpid());
7     int rc = fork();
8     if (rc < 0) {
9         // fork failed
10         fprintf(stderr, "fork failed\n");
11         exit(1);
12     } else if (rc == 0) {
13         // child (new process)
14         printf("hello, I am child (pid:%d)\n", (int) getpid());
15     } else {
16         // parent goes down this path (main)
17         printf("hello, I am parent of %d (pid:%d)\n",
18                rc, (int) getpid());
19     }
20     return 0;
21 }
22

```

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.

```

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

```

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

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

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

```

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

```

The CPU **scheduler**, a topic we'll discuss in great detail soon, determines which process runs at a given moment in time; because the scheduler is complex, we cannot usually make strong assumptions about what

it will choose to do, and hence which process will run first. This **non-determinism**, as it turns out, leads to some interesting problems, particularly in **multi-threaded programs**; hence, we'll see a lot more non-determinism when we study **concurrency** in the second part of the book.

5.2 The `wait()` System Call

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

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

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

(waiting for you to think and done)

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

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

With this code, we now know that the child will always print first. Why do we know that? Well, it might simply run first, as before, and thus print before the parent. However, if the parent does happen to run first, it will immediately call `wait()`; this system call won't return until the child has run and exited². 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 call³. This system call is useful when you want to run a program that is different from the calling program. For example, calling `fork()`

²There 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."

³On Linux, there are six variants of `exec()`: `execl()`, `execvp()`, `execle()`, `execv()`, `execvpe()`, and `execve()`. 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 main(int argc, char *argv[]) {
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        char *myargs[3];
16        myargs[0] = strdup("wc");    // program: "wc" (word count)
17        myargs[1] = strdup("p3.c");  // argument: file to count
18        myargs[2] = NULL;          // marks end of array
19        execvp(myargs[0], myargs); // runs word count
20        printf("this shouldn't print out");
21    } else {                // parent goes down this path (main)
22        int rc_wait = wait(NULL);
23        printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
24               rc, rc_wait, (int) getpid());
25    }
26    return 0;
27 }
```

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

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

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

```

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

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

TIP: GETTING IT RIGHT (LAMPSON'S LAW)

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

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

5.4 Why? Motivating The API

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

The shell is just a user program⁴. 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
```

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

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

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

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

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

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

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

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

```

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 main(int argc, char *argv[]) {
9     int rc = fork();
10    if (rc < 0) {
11        // fork failed
12        fprintf(stderr, "fork failed\n");
13        exit(1);
14    } else if (rc == 0) {
15        // 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");    // arg: file to count
23        myargs[2] = NULL;             // mark end of array
24        execvp(myargs[0], myargs);   // runs word count
25    } else {
26        // parent goes down this path (main)
27        int rc_wait = wait(NULL);
28    }
29    return 0;
30 }
```

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

5.5 Process Control And Users

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

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

ASIDE: RTFM — READ THE MAN PAGES

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

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

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

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

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

5.6 Useful Tools

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

ASIDE: THE SUPERUSER (ROOT)

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

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

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

5.7 Summary

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

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

ASIDE: KEY PROCESS API TERMS

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

References

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

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

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

[J16] “They could be twins!” by Phoebe Jackson-Edwards. The Daily Mail. March 1, 2016. Available: www.dailymail.co.uk/femail/article-3469189/Photos-children-look-IDENTICAL-parents-age-sweep-web.html. *This hard-hitting piece of journalism shows a bunch of weirdly similar child/parent photos and is frankly kind of mesmerizing. Go ahead, waste two minutes of your life and check it out. But don’t forget to come back here! This, in a microcosm, is the danger of surfing the web.*

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

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

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

ASIDE: CODING HOMEWORKS

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

Homework (Code)

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

Questions

1. Write a program that calls `fork()`. Before calling `fork()`, have the main process access a variable (e.g., `x`) and set its value to something (e.g., 100). What value is the variable in the child process? What happens to the variable when both the child and parent change the value of `x`?
2. Write a program that opens a file (with the `open()` system call) and then calls `fork()` to create a new process. Can both the child and parent access the file descriptor returned by `open()`? What happens when they are writing to the file concurrently, i.e., at the same time?
3. Write another program using `fork()`. The child process should print "hello"; the parent process should print "goodbye". You should try to ensure that the child process always prints first; can you do this *without* calling `wait()` in the parent?
4. Write a program that calls `fork()` and then calls some form of `exec()` to run the program `/bin/ls`. See if you can try all of the variants of `exec()`, including (on Linux) `exec1()`, `execle()`, `execlp()`, `execv()`, `execvp()`, and `execvpe()`. Why do you think there are so many variants of the same basic call?
5. Now write a program that uses `wait()` to wait for the child process to finish in the parent. What does `wait()` return? What happens if you use `wait()` in the child?

6. Write a slight modification of the previous program, this time using `waitpid()` instead of `wait()`. When would `waitpid()` be useful?
7. Write a program that creates a child process, and then in the child closes standard output (`STDOUT_FILENO`). What happens if the child calls `printf()` to print some output after closing the descriptor?
8. Write a program that creates two children, and connects the standard output of one to the standard input of the other, using the `pipe()` system call.

Mechanism of process execution

Low-level mechanisms

- How does the OS run a process?
- How does it handle a system call?
- How does it context switch from one process to the other?

Process Execution

- OS allocates memory and creates memory image
 - Code and data (from exe)
 - Stack and heap
- Points CPU program counter to current instruction
 - Other registers may store operands, return values etc.
- After setup, OS is out of the way and process executes directly on CPU

A simple function call

- A function call translates to a jump instruction
- A new stack frame pushed to stack and stack pointer (SP) updated
- Old value of PC (return value) pushed to stack and PC updated
- Stack frame contains return value, function arguments etc.

How is a system call different?

- CPU hardware has multiple privilege levels
 - One to run user code: user mode
 - One to run OS code like system calls: kernel mode
 - Some instructions execute only in kernel mode
- Kernel does not trust user stack
 - Uses a separate kernel stack when in kernel mode
- Kernel does not trust user provided addresses to jump to
 - Kernel sets up Interrupt Descriptor Table (IDT) at boot time
 - IDT has addresses of kernel functions to run for system calls and other events

Mechanism of system call: trap instruction

- When system call must be made, a special trap instruction is run (usually hidden from user by libc)
- Trap instruction execution
 - Move CPU to higher privilege level
 - Switch to kernel stack
 - Save context (old PC, registers) on kernel stack
 - Look up address in IDT and jump to trap handler function in OS code

More on the trap instruction

- Trap instruction is executed on hardware in following cases:
 - System call (program needs OSservice)
 - Program fault (program does something illegal, e.g., access memory it doesn't have access to)
 - Interrupt (external device needs attention of OS, e.g., a network packet has arrived on network card)
- Across all cases, the mechanism is: save context on kernel stack and switch to OSaddress in IDT
- IDT has many entries: which to use?
 - System calls/interrupts store a number in a CPUregister before calling trap, to identify which IDT entry to use

Return from trap

- When OS is done handling syscall or interrupt, it calls a special instruction return-from-trap
 - Restore context of CPU registers from kernel stack
 - Change CPU privilege from kernel mode to user mode
 - Restore PC and jump to user code after trap
- User process unaware that it was suspended, resumes execution as always
- Must you always return to the same user process from kernel mode? No
- Before returning to user mode, OS checks if it must switch to another process

Why switch between processes?

- Sometimes when OS is in kernel mode, it cannot return back to the same process it left
 - Process has exited or must be terminated (e.g., segfault)
 - Process has made a blocking system call
- Sometimes, the OS does not want to return back to the same process
 - The process has run for too long
 - Must timeshare CPU with other processes
- In such cases, OS performs a context switch to switch from one process to another

The OS scheduler

- OS scheduler has two parts
 - Policy to pick which process to run
 - Mechanism to switch to that process
- Non preemptive (cooperative) schedulers are polite
 - Switch only if process blocked or terminated
- Preemptive (non-cooperative) schedulers can switch even when process is ready to continue
 - CPU generates periodic timer interrupt
 - After servicing interrupt, OS checks if the current process has run for too long

r

Mechanism of context switch

- Example: process A has moved from user to kernel mode, OS decides it must switch from A to B
- Save context (PC, registers, kernel stack pointer) of A on kernel stack
- Switch SP to kernel stack of B
- Restore context from B's kernel stack
- Who has saved registers on B's kernel stack?
 - OS did, when it switched out B in the past
- Now, CPU is running B in kernel mode, return-from-trap to switch to user mode of B

A subtlety on saving context

- Context (PC and other CPU registers) saved on the kernel stack in two different scenarios
- When going from user mode to kernel mode, user context (e.g., which instruction of user code you stopped at) is saved on kernel stack by the trap instruction
 - Restored by return-from-trap
- During a context switch, kernel context (e.g., where you stopped in the OS code) of process A is saved on the kernel stack of A by the context switching code
 - Restores kernel context of process B

Mechanism: Limited Direct Execution

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

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

THE CRUX:

HOW TO EFFICIENTLY VIRTUALIZE THE CPU WITH CONTROL

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

6.1 Basic Technique: Limited Direct Execution

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

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

Figure 6.1: Direct Execution Protocol (Without Limits)

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

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

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

6.2 Problem #1: Restricted Operations

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

THE CRUX: HOW TO PERFORM RESTRICTED OPERATIONS

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

ASIDE: WHY SYSTEM CALLS LOOK LIKE PROCEDURE CALLS

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

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

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

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

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

TIP: USE PROTECTED CONTROL TRANSFER

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

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

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

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

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

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

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

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

Figure 6.2: Limited Direct Execution Protocol

locations of these **trap handlers**, usually with some kind of special instruction. Once the hardware is informed, it remembers the location of these handlers until the machine is next rebooted, and thus the hardware knows what to do (i.e., what code to jump to) when system calls and other exceptional events take place.

TIP: BE WARY OF USER INPUTS IN SECURE SYSTEMS

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

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

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

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

One last aside: being able to execute the instruction to tell the hardware where the trap tables are is a very powerful capability. Thus, as you might have guessed, it is also a **privileged** operation. If you try to execute this instruction in user mode, the hardware won’t let you, and you can probably guess what will happen (hint: adios, offending program). Point to ponder: what horrible things could you do to a system if you could install your own trap table? Could you take over the machine?

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

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

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

6.3 Problem #2: Switching Between Processes

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

THE CRUX: HOW TO REGAIN CONTROL OF THE CPU

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

A Cooperative Approach: Wait For System Calls

One approach that some systems have taken in the past (for example, early versions of the Macintosh operating system [M11], or the old Xerox Alto system [A79]) is known as the **cooperative** approach. In this style, the OS *trusts* the processes of the system to behave reasonably. Processes that run for too long are assumed to periodically give up the CPU so that the OS can decide to run some other task.

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

Applications also transfer control to the OS when they do something illegal. For example, if an application divides by zero, or tries to access memory that it shouldn't be able to access, it will generate a **trap** to the

OS. The OS will then have control of the CPU again (and likely terminate the offending process).

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

A Non-Cooperative Approach: The OS Takes Control

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

THE CRUX: HOW TO GAIN CONTROL WITHOUT COOPERATION

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

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

As we discussed before with system calls, the OS must inform the hardware of which code to run when the timer interrupt occurs; thus, at boot time, the OS does exactly that. Second, also during the boot sequence, the OS must start the timer, which is of course a privileged

TIP: DEALING WITH APPLICATION MISBEHAVIOR

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

operation. Once the timer has begun, the OS can thus feel safe in that control will eventually be returned to it, and thus the OS is free to run user programs. The timer can also be turned off (also a privileged operation), something we will discuss later when we understand concurrency in more detail.

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

Saving and Restoring Context

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

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

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

TIP: USE THE TIMER INTERRUPT TO REGAIN CONTROL

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

TIP: REBOOT IS USEFUL

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

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

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

the soon-to-be-executing process becomes the currently-running process. And thus the context switch is complete.

A timeline of the entire process is shown in Figure 6.3. In this example, Process A is running and then is interrupted by the timer interrupt. The hardware saves its registers (onto its kernel stack) and enters the kernel (switching to kernel mode). In the timer interrupt handler, the OS decides to switch from running Process A to Process B. At that point, it calls the `switch()` routine, which carefully saves current register values (into the process structure of A), restores the registers of Process B (from its process structure entry), and then **switches contexts**, specifically by changing the stack pointer to use B's kernel stack (and not A's). Finally, the OS returns-from-trap, which restores B's registers and starts running it.

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

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

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

Figure 6.3: Limited Direct Execution Protocol (Timer Interrupt)

6.4 Worried About Concurrency?

Some of you, as attentive and thoughtful readers, may be now thinking: “Hmm... what happens when, during a system call, a timer interrupt occurs?” or “What happens when you’re handling one interrupt and another one happens? Doesn’t that get hard to handle in the kernel?” Good questions — we really have some hope for you yet!

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

To whet your appetite, we’ll just sketch some basics of how the OS handles these tricky situations. One simple thing an OS might do is **disable interrupts** during interrupt processing; doing so ensures that when

```

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

one interrupt is being handled, no other one will be delivered to the CPU. Of course, the OS has to be careful in doing so; disabling interrupts for too long could lead to lost interrupts, which is (in technical terms) bad.

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

6.5 Summary

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

ASIDE: HOW LONG CONTEXT SWITCHES TAKE

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

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

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

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

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

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

ASIDE: KEY CPU VIRTUALIZATION TERMS (MECHANISMS)

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

References

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

Homework (Measurement)

ASIDE: MEASUREMENT HOMEWORKS

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

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

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

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

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

Scheduling Policies

What is a scheduling policy?

- On context switch, which process to run next, from set of ready processes?
- OSscheduler schedules the CPUrequests (bursts) of processes
 - CPUburst = the CPUtime used by a process in a continuous stretch
 - If a process comes back after I/O wait, it counts as a fresh CPU burst

What are we trying to optimize?

- Maximize (utilization = fraction of time CPU is used)
- Minimize average (turnaround time = time from process arrival to completion)
- Minimize average (response time = time from process arrival to first scheduling)
- Fairness: all processes must be treated equally
- Minimize overhead: run process long enough to amortize cost of context switch (~1 microsecond)

First-In-First-Out (FIFO)

- Example: three processes arrive at $t=0$ in the order A,B,C
- Problem: convoy effect
- Turnaround times tend to be high

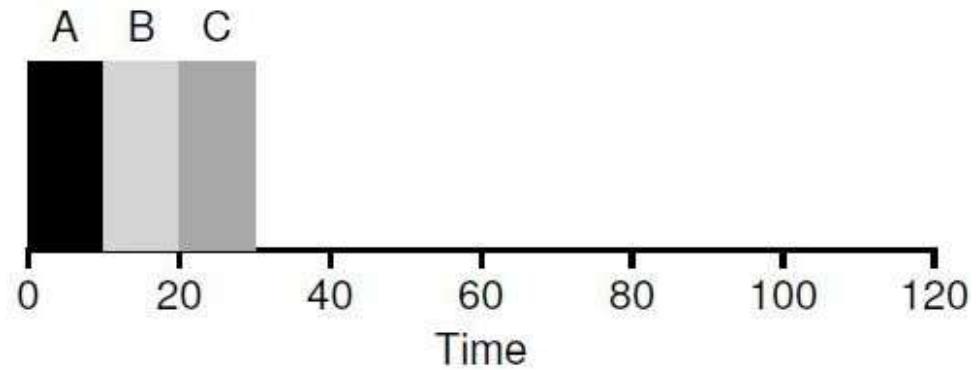


Figure 7.1: FIFO Simple Example

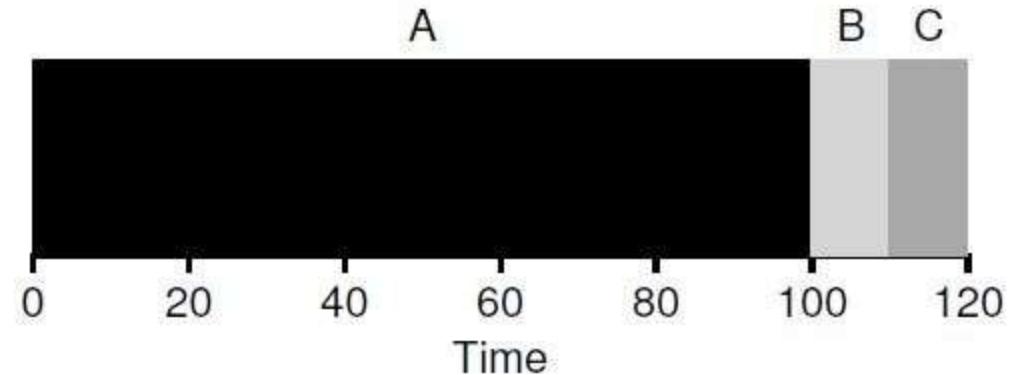


Figure 7.2: Why FIFO Is Not That Great

Shortest Job First (SJF)

- Provably optimal when all processes arrive together.
- SJF is non-preemptive, so short jobs can still get stuck behind long ones.

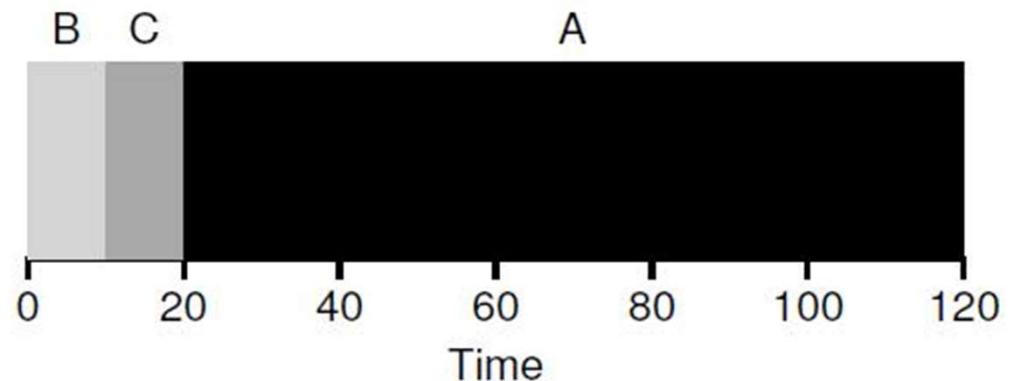


Figure 7.3: SJF Simple Example

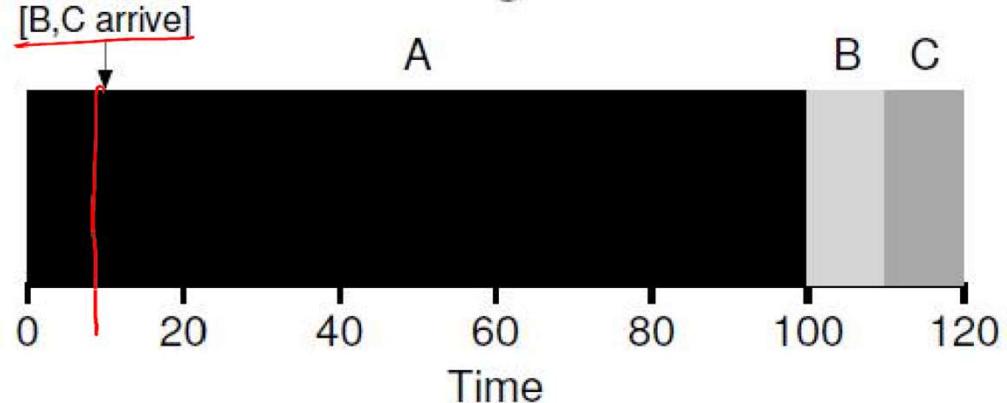


Figure 7.4: SJF With Late Arrivals From B and C

Shortest Time-to-Completion First (STCF)

- Also called Shortest Remaining Time First (SRTF)
- Preemptive scheduler
- Preempts running task if time left is more than that of new arrival

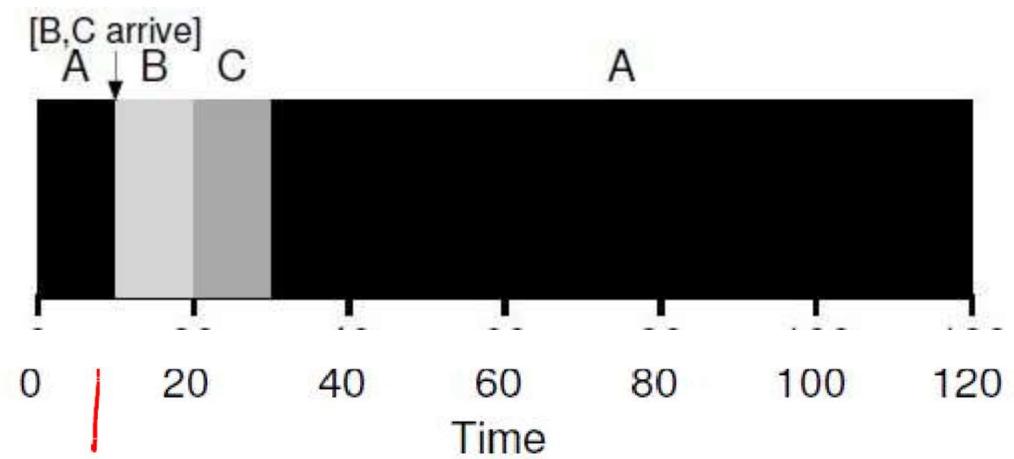


Figure 7.5: STCF Simple Example

Round Robin (RR)

- Every process executes for a fixed quantum slice
- Slice big enough to amortize cost of context switch
- Preemptive
- Good for response time and fairness
- Bad for turnaround time

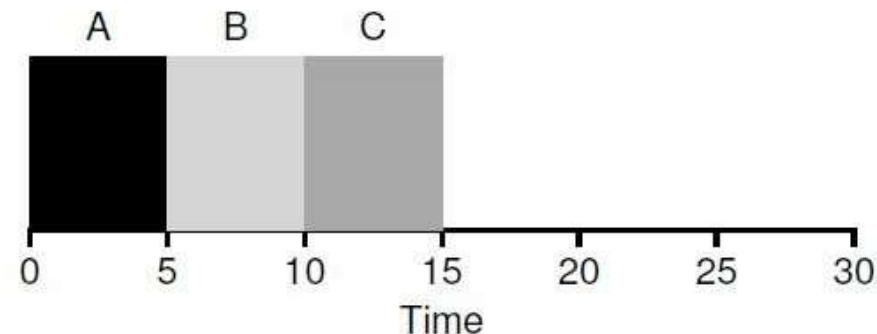


Figure 7.6: SJF Again (Bad for Response Time)

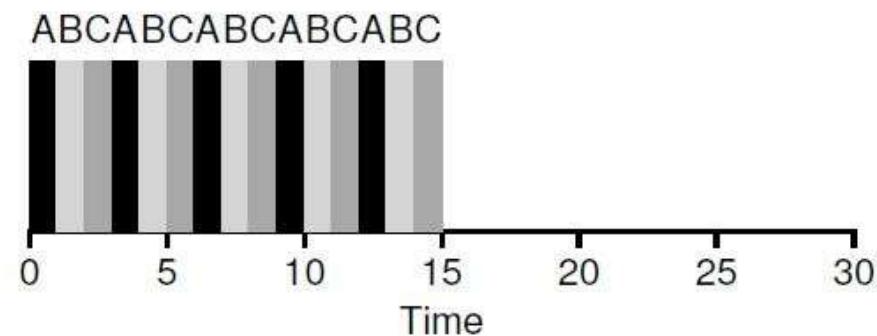


Figure 7.7: Round Robin (Good for Response Time)

Schedulers in real systems

- Real schedulers are more complex
- For example, Linux uses a Multi Level Feedback Queue (MLFQ)
 - Many queues, in order of priority
 - Process from highest priority queue scheduled first
 - Within same priority, any algorithm like RR
 - Priority of process decays with its age

Scheduling: Introduction

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

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

THE CRUX: HOW TO DEVELOP SCHEDULING POLICY

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

7.1 Workload Assumptions

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

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

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

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

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

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

7.2 Scheduling Metrics

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

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

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

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)**.

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

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

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

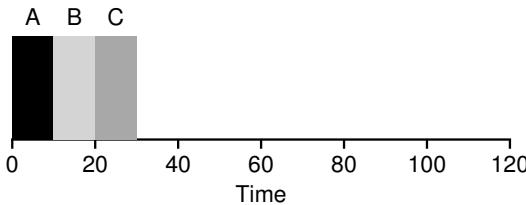


Figure 7.1: FIFO Simple Example

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

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

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

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

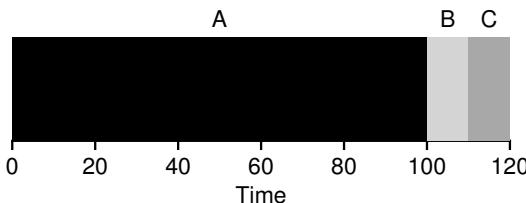


Figure 7.2: Why FIFO Is Not That Great

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

This problem is generally referred to as the **convoy effect** [B+79], where a number of relatively-short potential consumers of a resource get queued

TIP: THE PRINCIPLE OF SJF

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

behind a heavyweight resource consumer. This scheduling scenario might remind you of a single line at a grocery store and what you feel like when you see the person in front of you with three carts full of provisions and a checkbook out; it’s going to be a while².

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.

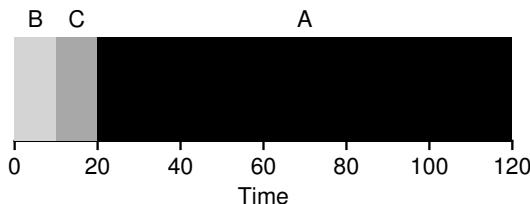


Figure 7.3: SJF Simple Example

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.

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

ASIDE: PREEMPTIVE SCHEDULERS

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

In fact, given our assumptions about jobs all arriving at the same time, we could prove that SJF is indeed an **optimal** scheduling algorithm. However, you are in a systems class, not theory or operations research; no proofs are allowed.

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

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

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

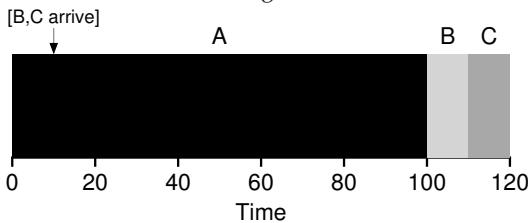


Figure 7.4: SJF With Late Arrivals From B and C

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

7.5 Shortest Time-to-Completion First (STCF)

To address this concern, we need to relax assumption 3 (that jobs must run to completion), so let's do that. We also need some machinery within the scheduler itself. As you might have guessed, given our previous discussion about timer interrupts and context switching, the scheduler can

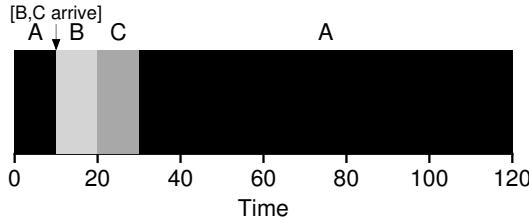


Figure 7.5: STCF Simple Example

certainly do something else when B and C arrive: it can **preempt** job A and decide to run another job, perhaps continuing A later. SJF by our definition is a **non-preemptive** scheduler, and thus suffers from the problems described above.

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

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

7.6 A New Metric: Response Time

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

We define response time as the time from when the job arrives in a system to the first time it is scheduled³. More formally:

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

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

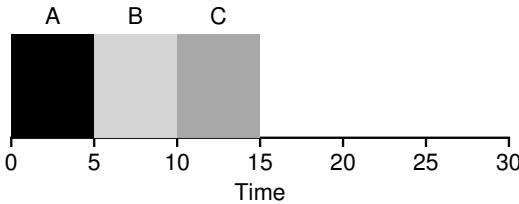


Figure 7.6: SJF Again (Bad for Response Time)

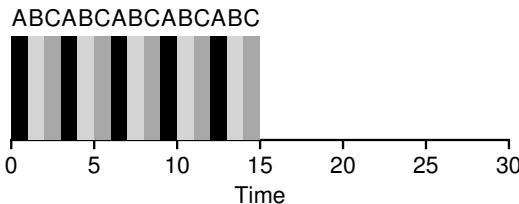


Figure 7.7: Round Robin (Good For Response Time)

For example, if we had the schedule 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 to see a response from the system just because some other job got scheduled in front of yours: not too pleasant.

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

7.7 Round Robin

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

To understand RR in more detail, let's look at an example. Assume three jobs A, B, and C arrive at the same time in the system, and that

TIP: AMORTIZATION CAN REDUCE COSTS

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

they each wish to run for 5 seconds. An SJF scheduler runs each job to completion before running another (Figure 7.6). In contrast, RR with a time-slice of 1 second would cycle through the jobs quickly (Figure 7.7).

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

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

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

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

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

More generally, any policy (such as RR) that is **fair**, i.e., that evenly divides the CPU among active processes on a small time scale, will perform poorly on metrics such as turnaround time. Indeed, this is an inherent trade-off: if you are willing to be unfair, you can run shorter jobs to completion, but at the cost of response time; if you instead value fairness,

TIP: OVERLAP ENABLES HIGHER UTILIZATION

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

response time is lowered, but at the cost of turnaround time. This type of **trade-off** is common in systems; you can't have your cake and eat it too⁴.

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

7.8 Incorporating I/O

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

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

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

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

Assume we are trying to build a STCF scheduler. How should such a scheduler account for the fact that A is broken up into 5 10-ms sub-jobs,

⁴A saying that confuses people, because it should be "You can't *keep* your cake and eat it too" (which is kind of obvious, no?). Amazingly, there is a wikipedia page about this saying; even more amazingly, it is kind of fun to read [W15]. As they say in Italian, you can't *Avere la botte piena e la moglie ubriaca*.

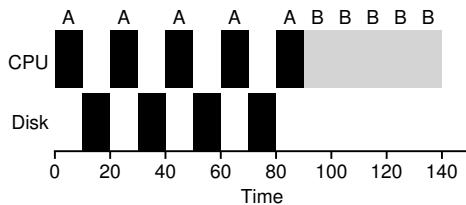


Figure 7.8: Poor Use Of Resources

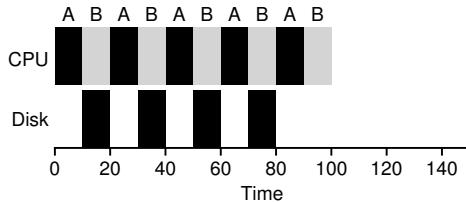


Figure 7.9: Overlap Allows Better Use Of Resources

whereas B is just a single 50-ms CPU demand? Clearly, just running one job and then the other without considering how to take I/O into account makes little sense.

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

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

7.9 No More Oracle

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

7.10 Summary

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

References

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

Homework (Simulation)

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

Questions

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

Scheduling Policies

What is a scheduling policy?

- On context switch, which process to run next, from set of ready processes?
- OSscheduler schedules the CPUrequests (bursts) of processes
 - CPUburst = the CPUtime used by a process in a continuous stretch
 - If a process comes back after I/O wait, it counts as a fresh CPU burst

What are we trying to optimize?

- Maximize (utilization = fraction of time CPU is used)
- Minimize average (turnaround time = time from process arrival to completion)
- Minimize average (response time = time from process arrival to first scheduling)
- Fairness: all processes must be treated equally
- Minimize overhead: run process long enough to amortize cost of context switch (~1 microsecond)

First-In-First-Out (FIFO)

- Example: three processes arrive at $t=0$ in the order A,B,C
- Problem: convoy effect
- Turnaround times tend to be high

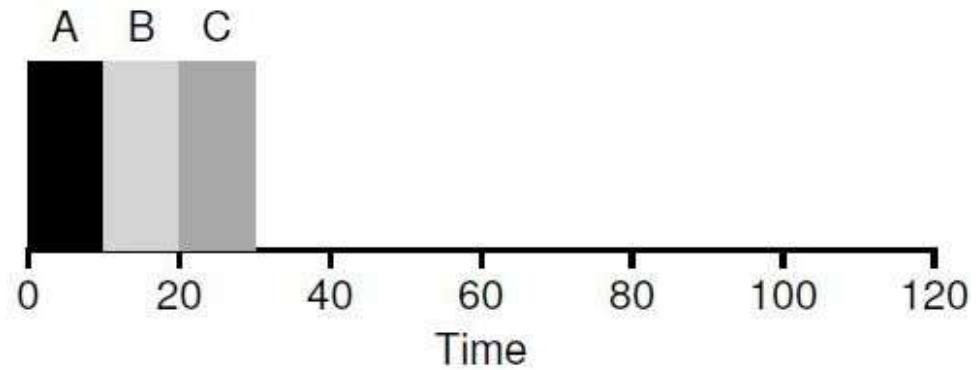


Figure 7.1: FIFO Simple Example

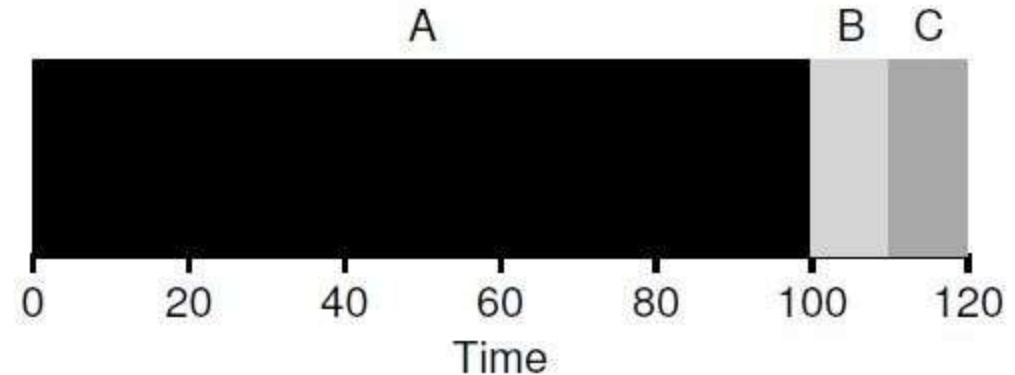


Figure 7.2: Why FIFO Is Not That Great

Shortest Job First (SJF)

- Provably optimal when all processes arrive together.
- SJF is non-preemptive, so short jobs can still get stuck behind long ones.

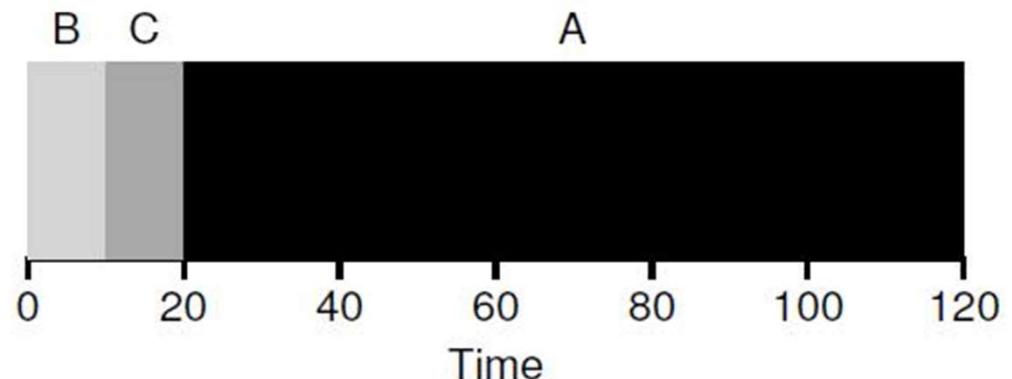


Figure 7.3: SJF Simple Example

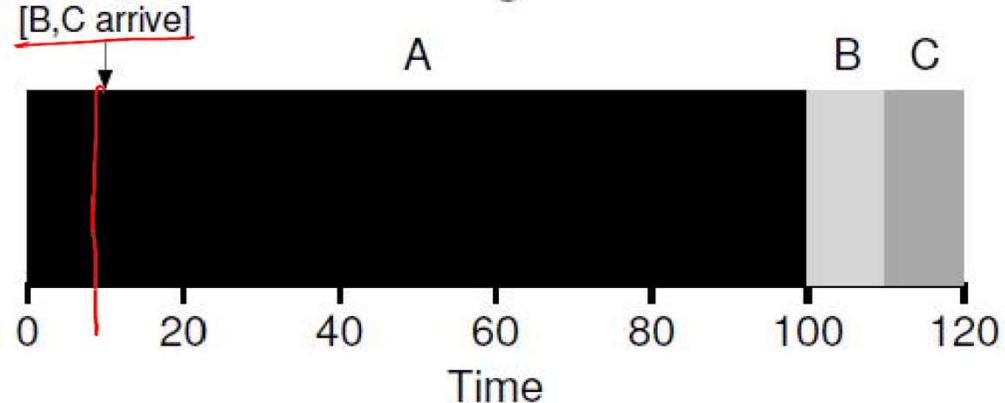


Figure 7.4: SJF With Late Arrivals From B and C

Shortest Time-to-Completion First (STCF)

- Also called Shortest Remaining Time First (SRTF)
- Preemptive scheduler
- Preempts running task if time left is more than that of new arrival

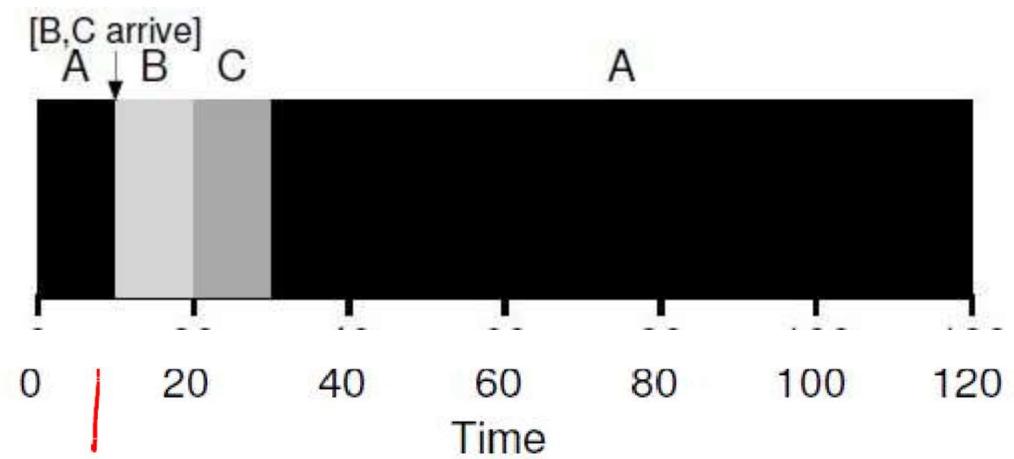


Figure 7.5: STCF Simple Example

Round Robin (RR)

- Every process executes for a fixed quantum slice
- Slice big enough to amortize cost of context switch
- Preemptive
- Good for response time and fairness
- Bad for turnaround time

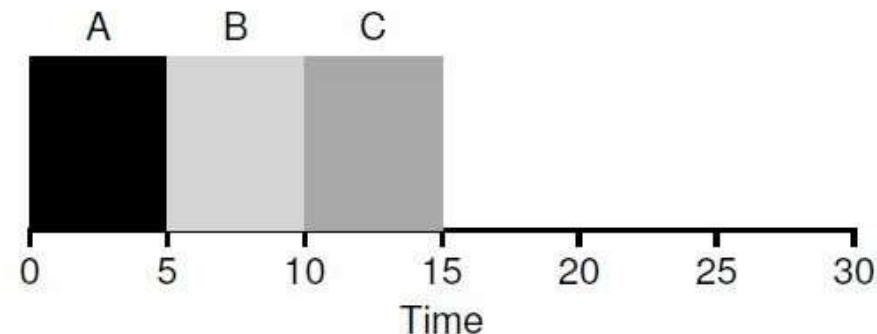


Figure 7.6: SJF Again (Bad for Response Time)

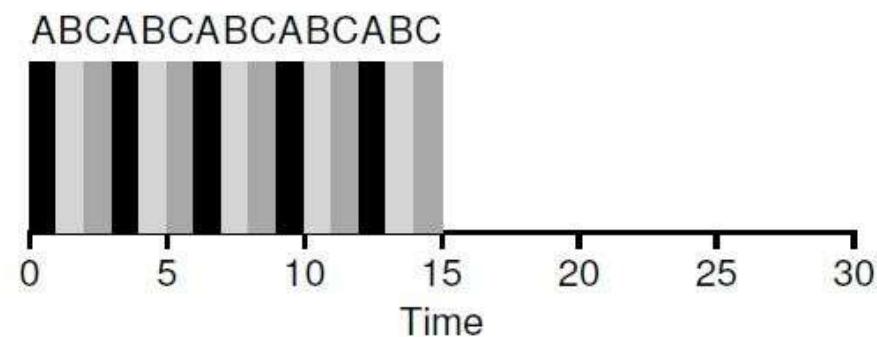


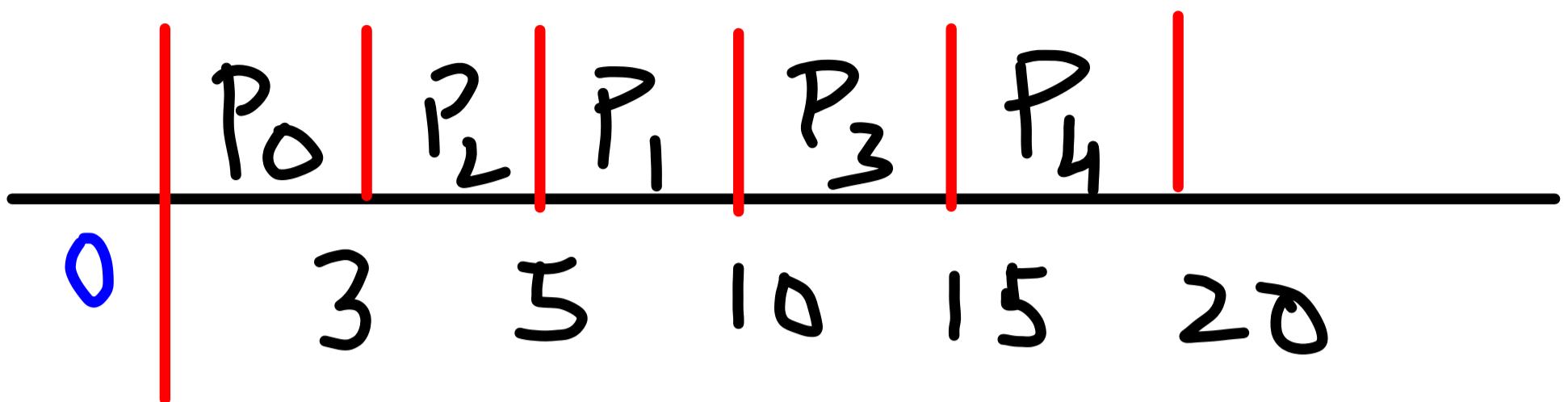
Figure 7.7: Round Robin (Good for Response Time)

Schedulers in real systems

- Real schedulers are more complex
- For example, Linux uses a Multi Level Feedback Queue (MLFQ)
 - Many queues, in order of priority
 - Process from highest priority queue scheduled first
 - Within same priority, any algorithm like RR
 - Priority of process decays with its age

5] F (non-Pre-emptive)

Process	A7	B7
P ₀	0	3
P ₁	1	5
P ₂	3	2
P ₃	9	5
P ₄	12	5



Formula

$$TT = CT - AT$$

$$WT = TT - BT$$

$RST = \text{First instance} - AT \text{ of Process}$

ρ_1 ρ_3 ρ_2 ρ ρ_0 ρ

$\bar{\rho}$ ρ $\omega - \delta$ AT

γ γ γ γ γ BT

ρ_0 $\bar{\gamma}$ γ \bar{o} ω CT

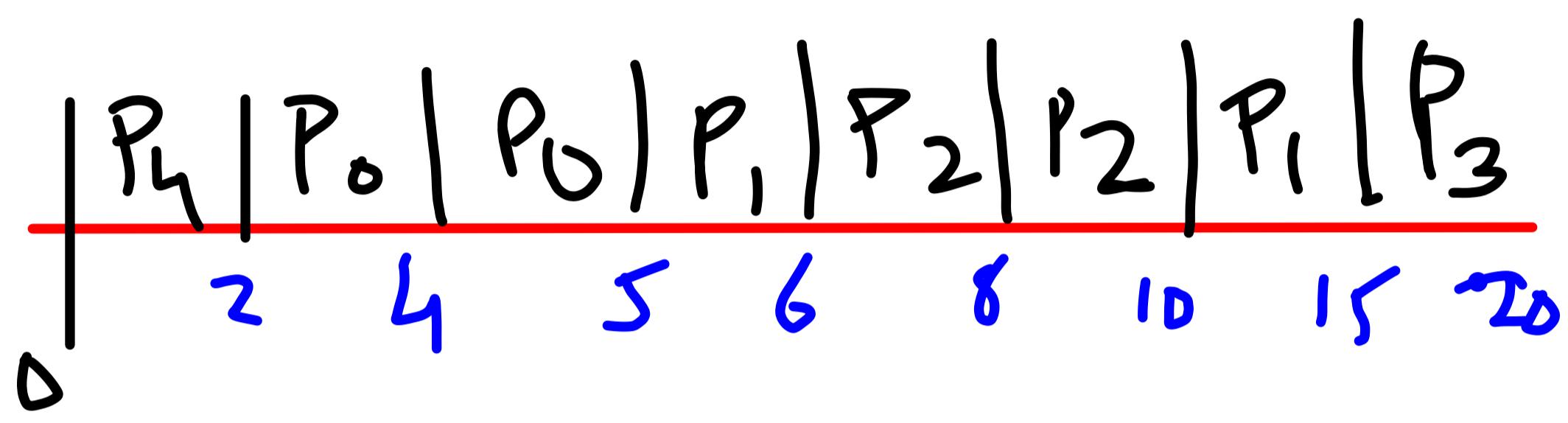
5.6m | 8 6 2 9 ω TAT

1.6m | 3 - 0 H 0 W T

ω - 0 H 0 RST

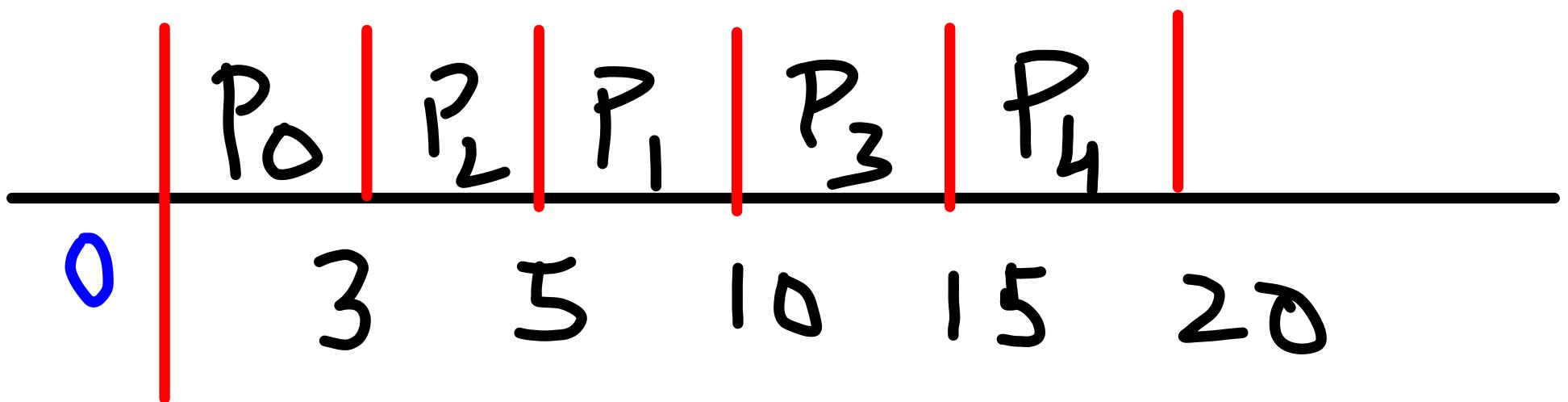
SJF PRE-EMPTIVE

PROCESS	AT	BT
P ₀	2	3
P ₁	4	6
P ₂	6	4
P ₃	8	5
P ₄	0	2



5] F (non-Pre-emptive)

Process	A7	B7
P ₀	0	3
P ₁	1	5
P ₂	3	2
P ₃	9	5
P ₄	12	5



Formula

$$TT = CT - AT$$

$$WT = TT - BT$$

$RST = \text{First instance} -$
 $AT \text{ of Process}$

ρ_1 ρ_3 ρ_2 ρ_1 ρ_0 ρ

$\bar{\rho}$ ρ $\omega - \delta$

γ γ γ γ γ

ρ_0 $\bar{\gamma}$ γ \bar{o} w

AT

BT

CT

TAT

WT

RST

5.6m
1.6m

γ - COHO

γ - COHO

SJF PRE-EMPTIVE

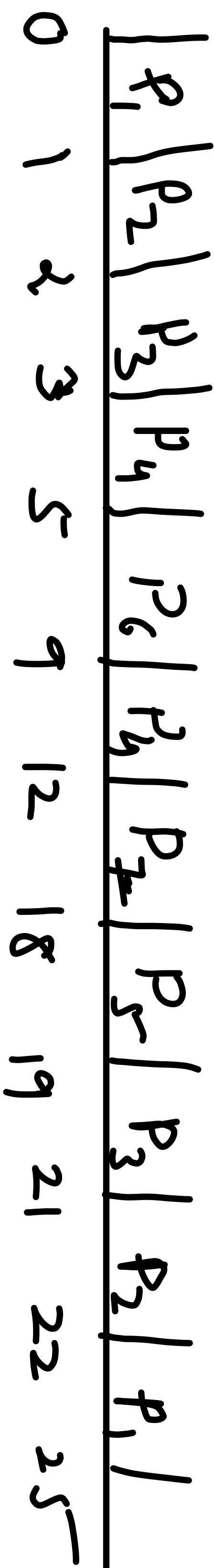
PROCESS	AT	BT
P ₀	2	3
P ₁	4	6
P ₂	6	4
P ₃	8	5
P ₄	0	2

	P ₁	P ₀	P ₅	P ₁	P ₂	P ₂	P ₁	P ₃
D	2	4	5	6	8	10	15	20
P ₀	2	3	5	3	0	0	0	0
P ₁	4	6	15	11	5	1	1	1
P ₂	6	4	10	4	0	0	0	0
P ₃	8	5	20	12	7	7	7	7
P ₄	0	2	2	2	0	0	0	0
				$\frac{32}{15} = \frac{12}{5} = 2.4 \text{ ms}$				
				6.4 ms				

Priority Scheduling

- Preemptive mode

P	Priority	AT	BT
1	2(L)	0	4
2	4	1	2
3	6	2	3
4	10	3	5
5	8	4	1
6	12(H)	5	4
7	9	6	6



at $t=9$ all process
arrived to the system

p_1 p_2 p_3 p_4 p_5 p_7

Priority 2 4 6 10 8 9

RBT

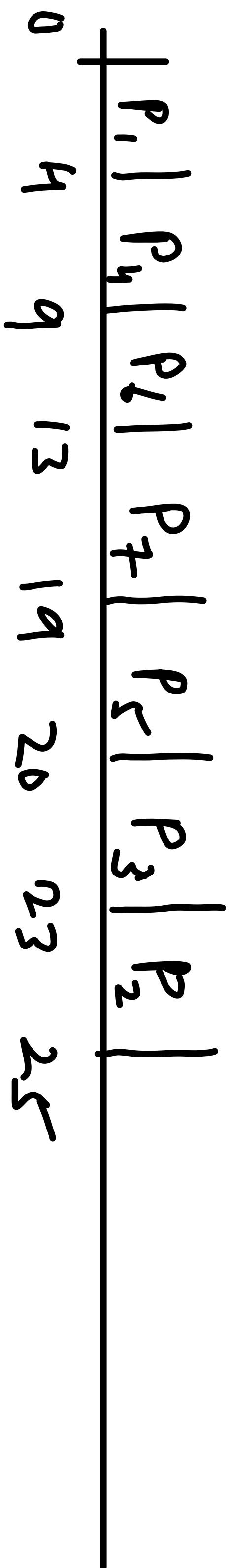
- 3 - 1 - 2 - 3 - 1

6

P	Priority	A/T	B/T	C/T	T/T	W/T	R/S/T
P_7	9	12	8	10	6	4	2
P_5	6	5	4	3	2	1	0
P_4	4	-	5	3	2	4	2.5
P_3	18	9	11	12	21	25	25
P_2	12	5	15	9	19	21	21
P_1	6	0	14	5	16	19	19
P_0	6	0	14	0	0	0	0

NON-PRE-EMPTIVE.

P	Priority	AT	BT
P ₁	2(L)	0	4
P ₂	4	1	2
P ₃	6	2	3
P ₄	10	3	5
P ₅	8	4	1
P ₆	12(H)	5	4
P ₇	9	6	6



$t = 0$

the time $t = 0 - 4$

$P_2 \quad P_3 \quad P_5 \quad P_6$

$P_2 \quad P_3 \quad P_4 \quad P_6$

$P_5 \quad P_6 \quad P_7 \quad P_8$

$P_5 \quad P_6 \quad P_7 \quad P_8$

$P_1 \quad P_4 \quad P_7 \quad P_8$

$P_1 \quad P_4 \quad P_7 \quad P_8$

$P_2 \quad P_3 \quad P_5 \quad P_6$

$P_2 \quad P_3 \quad P_4 \quad P_6$

RR

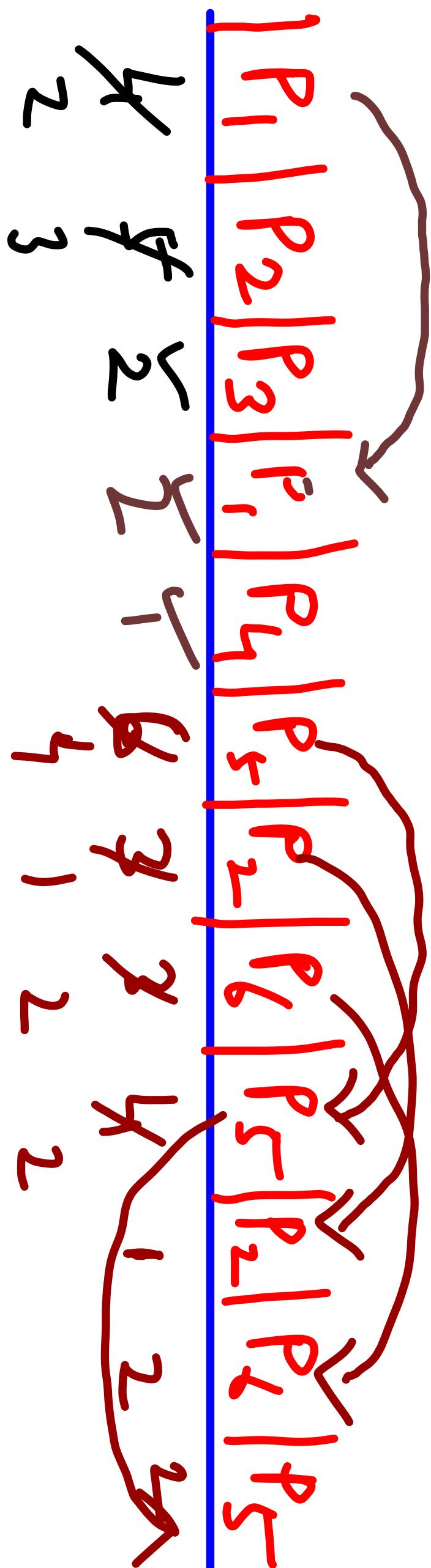
P	AT	BT
1	0	4
2	1	5
3	2	2
4	3	1
5	4	6
6	6	3

$$\overline{T}_0 = 2 \text{ ms}$$

each process will get
2ms to execute instruction

$P_1 | P_2 | P_3 | P_4 | P_5 | P_6 | P_7 | P_8 | P_9 |$
0 2 4 6 8 9 1 11 13 15 17 18 19 21

b/w 0-2 (up wave, $P_2 P_3$) Need to
maintain a queue for RR transition



T

AT

BT

CT

TAT

WT

RST

6 7 5 3 2 -

6 4 3 2 - 0

3 6 - 2 5 4

19 21 9 6 8

13 17 6 4 17 8

10 = 5 2 12 4

+ 4 5 2 - 0

Inter Process Communication (IPC)

Inter Process Communication (IPC)

- Processes do not share any memory with each other
- Some processes might want to work together for a task, so need to communicate information
- IPC mechanisms to share information between processes

Shared Memory

- Processes can both access same region of memory via `shmget()` system call
 - `int shmget (key_t key, int size, int shmflg)`
- By providing same key, two processes can get same segment of memory
- Can read/write to memory to communicate
- Need to take care that one is not overwriting other's data: how?

Signals

- A certain set of signals supported by OS
 - Some signals have fixed meaning (e.g., signal to terminate process)
 - Some signals can be user-defined
- Signals can be sent to a process by OS or another process (e.g., if you type Ctrl+C, OS sends SIGINT signal to running process)
- Signal handler: every process has a default code to execute for each signal
 - Exit on terminate signal
- Some signal handlers can be overridden to do other things

Sockets

- Sockets can be used for two processes on same machine or different machines to communicate
 - TCP/UDP sockets across machines
 - Unix sockets in local machine
- Communicating with sockets
 - Processes open sockets and connect them to each other
 - Messages written into one socket can be read from another
 - OS transfers data across socket buffers

Pipes

- Pipe system call returns two file descriptors
 - Read handle and write handle
 - A pipe is a half-duplex communication
 - Data written in one file descriptor can be read through another
- Regular pipes: both fd are in same process (how it is useful?)
 - Parent and child share fd after fork
 - Parent uses one end and child uses other end
- Named pipes: two endpoints of a pipe can be in different processes
- Pipe data buffered in OS buffers between write and read

Message Queues

- Mailbox abstraction
- Process can open a mailbox at a specified location
- Processes can send/receive messages from mailbox
- OS buffers messages between send and receive

Blocking vs. non-blocking communication

- Some IPC actions can block
 - Reading from socket/pipe that has no data, or reading from empty message queue
 - Writing to a full socket/pipe/message queue
- The system calls to read/write have versions that block or can return with an error code in case of failure
 - A socket read can return error indicating no data to be read, instead of blocking

Introduction to virtual memory

Why virtualize memory?

- Because real view of memory is messy!
- Earlier, memory had only code of one running process (and OS code)
- Now, multiple active processes timeshare CPU
 - Memory of many processes must be in memory
 - Non-contiguous too
- Need to hide this complexity from user

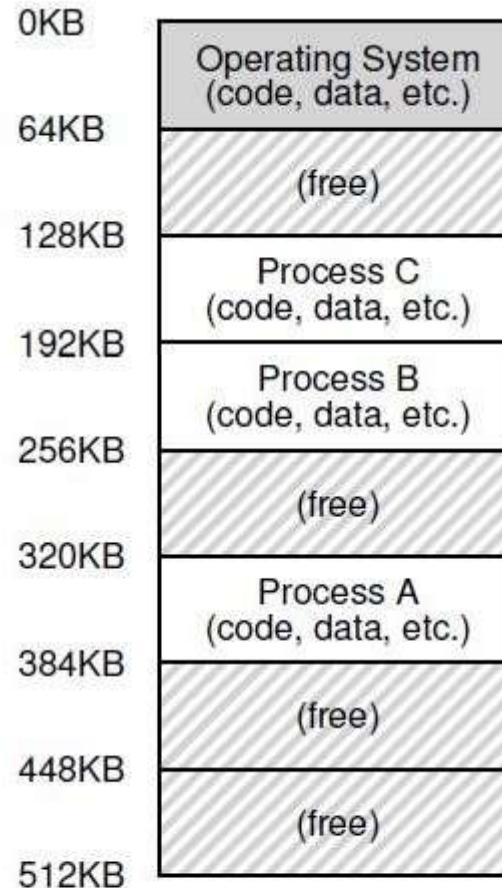


Figure 13.2: Three Processes: Sharing Memory

Abstraction: (Virtual) Address Space

- Virtual address space: every process assumes it has access to a large space of memory from address 0 to a MAX
- Contains program code (and static data), heap (dynamic allocations), and stack (used during function calls)
 - Stack and heap grow during runtime
- CPU issues loads and stores to virtual addresses

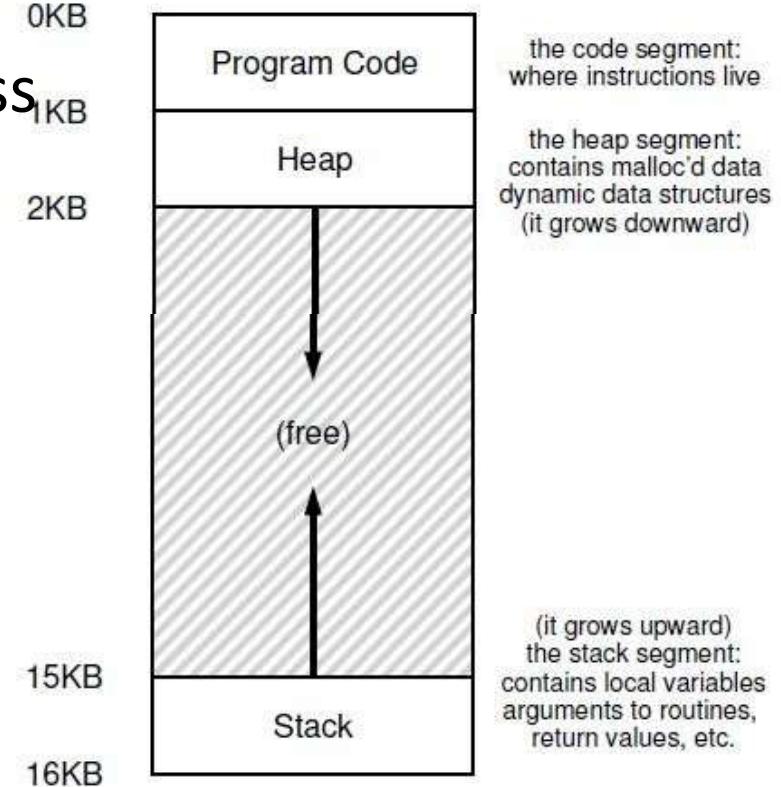
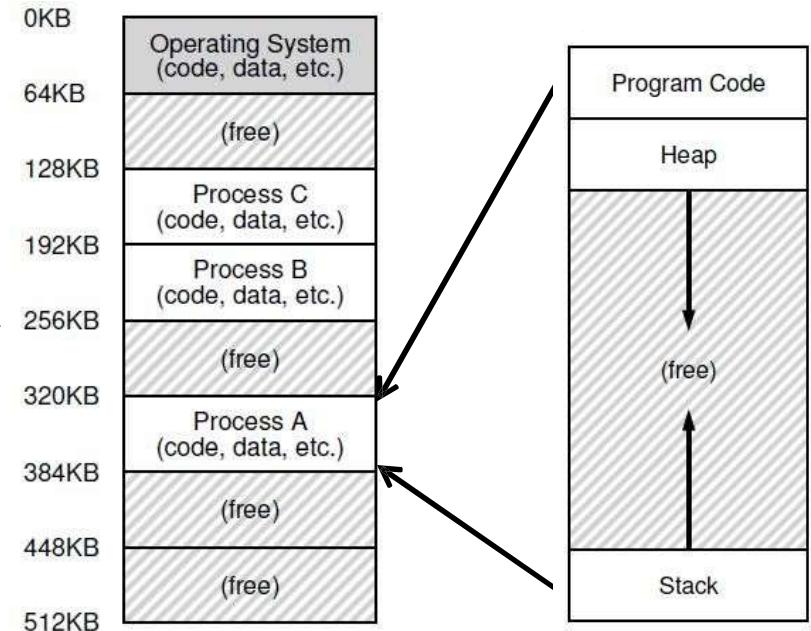


Figure 13.3: An Example Address Space

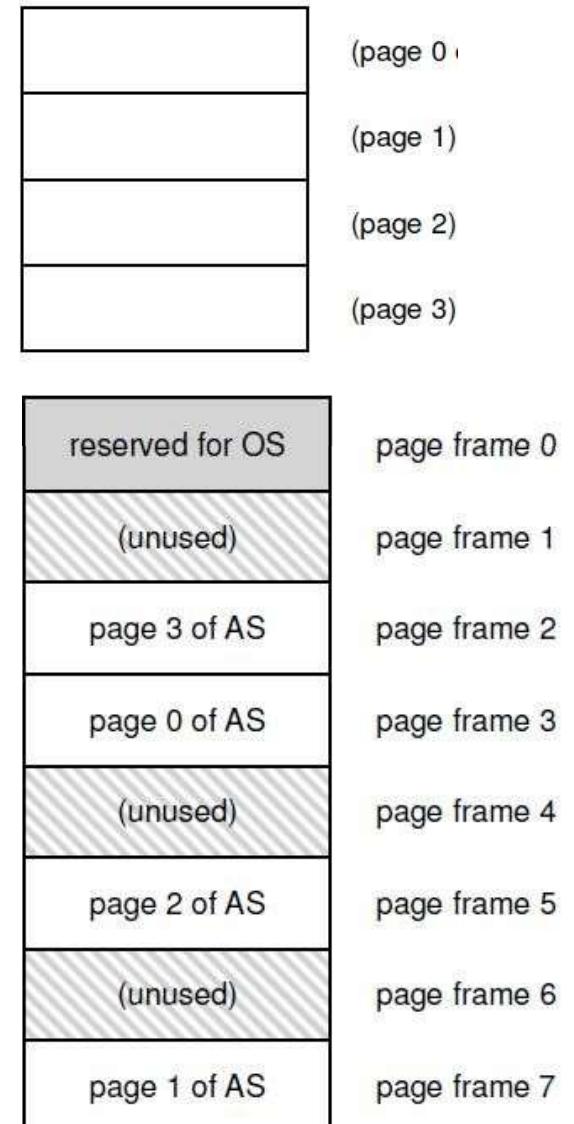
How is actual memory reached?

- Address translation from virtual addresses (VA) to physical addresses (PA)
 - CPU issues loads/stores to VA but memory hardware accesses PA
- OS allocates memory and tracks location of processes
- Translation done by memory hardware called Memory Management Unit (MMU)
 - OS makes the necessary information available



Example: Paging

- OS divides virtual address space into fixed size pages, physical memory into frames
- To allocate memory, a page is mapped to a free physical frame
- Page table stores mappings from virtual page number to physical frame number for a process (e.g, page 0 to frame 3)
- MMU has access to page tables, and uses it to translate VA to PA



Goals of memory virtualization

- Transparency: user programs should not be aware of the messy details
- Efficiency: minimize overhead and wastage in terms of memory space and access time
- Isolation and protection: a user process should not be able to access anything outside its address space

How can a user allocate memory?

- OS allocates a set of pages to the memory image of the process
- Within this image
 - Static/global variables are allocated in the executable
 - Local variables of a function on stack
 - Dynamic allocation with malloc on the heap

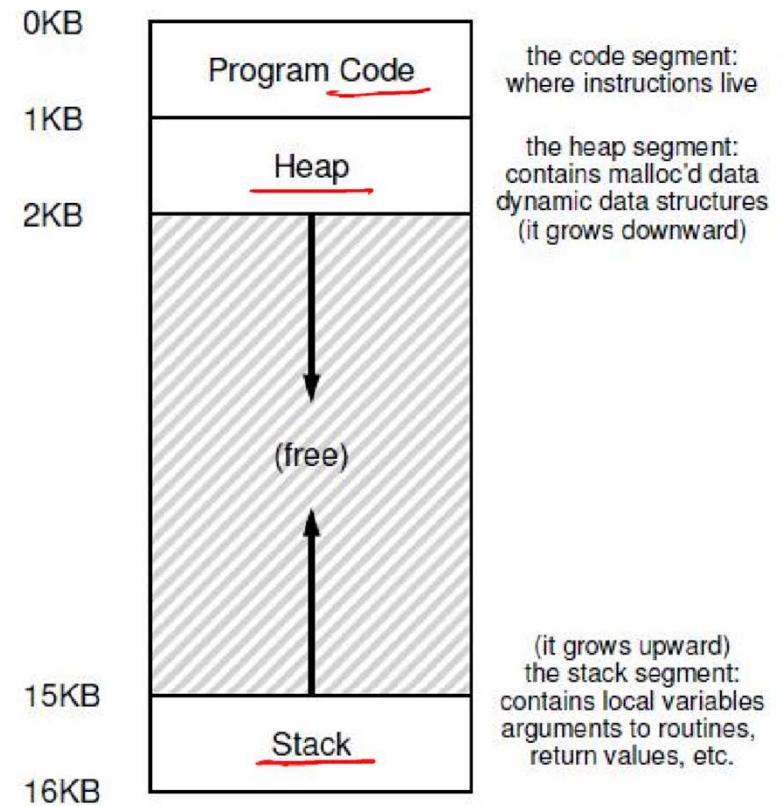
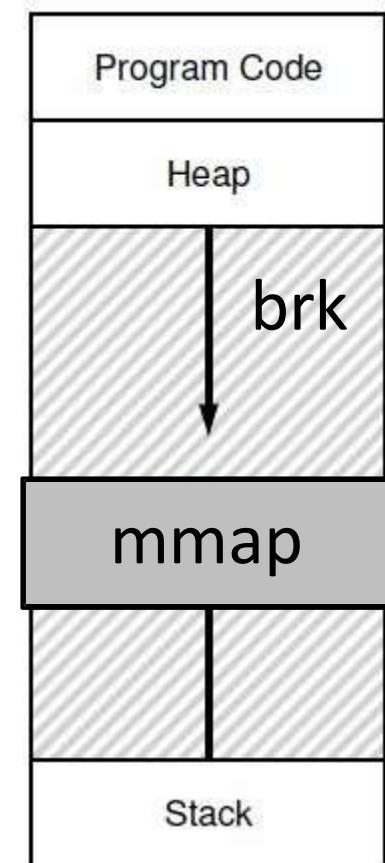


Figure 13.3: An Example Address Space

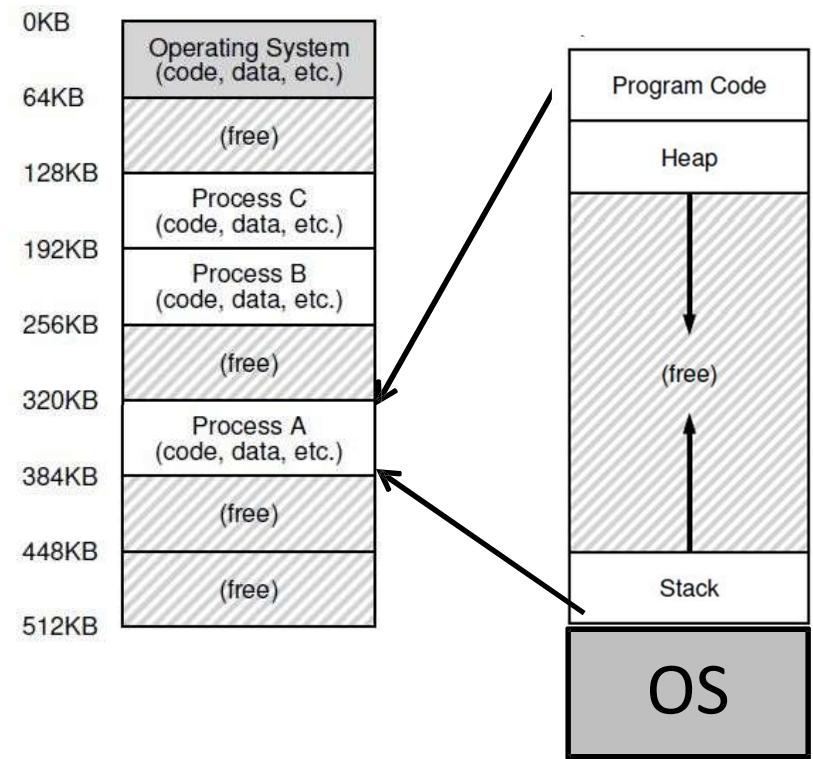
Memory allocation system calls

- `malloc` implemented by C library
 - Algorithms for efficient memory allocation and free space management
- To grow heap, libc uses the `brk`/`sbrk` system call
- A program can also allocate a page sized memory using the `mmap()` system call
 - Gets “anonymous” page from OS



A subtle point: what is the address space of the OS?

- OS is not a separate process with its own address space
- Instead, OS code is part of the address space of every process
- A process sees OS as part of its code (e.g., library)
- Page tables map the OS addresses to OS code



How does the OS allocate memory?

- OS needs memory for its data structures
- For large allocations, OS allocates a page
- For smaller allocations, OS uses various memory allocation algorithms (more later)
 - Cannot use libc and malloc in kernel!

The Abstraction: Address Spaces

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

13.1 Early Systems

From the perspective of memory, early machines didn't provide much of an abstraction to users. Basically, the physical memory of the machine looked something like what you see in Figure 13.1 (page 2).

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

13.2 Multiprogramming and Time Sharing

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

Soon enough, however, people began demanding more of machines, and the era of **time sharing** was born [S59, L60, M62, M83]. Specifically, many realized the limitations of batch computing, particularly on programmers themselves [CV65], who were tired of long (and hence ineffec-

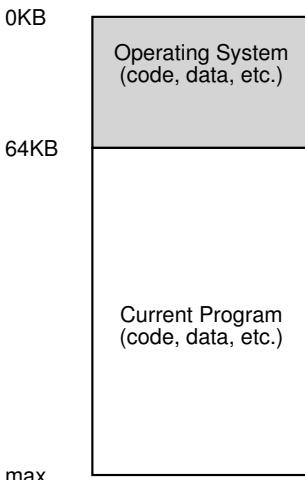


Figure 13.1: **Operating Systems: The Early Days**

tive) program-debug cycles. The notion of **interactivity** became important, as many users might be concurrently using a machine, each waiting for (or hoping for) a timely response from their currently-executing tasks.

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

Unfortunately, this approach has a big problem: it is way too slow, particularly as memory grows. While saving and restoring register-level state (the PC, general-purpose registers, etc.) is relatively fast, saving the entire contents of memory to disk is brutally non-performant. Thus, what we'd rather do is leave processes in memory while switching between them, allowing the OS to implement time sharing efficiently (as shown in Figure 13.2, page 3).

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

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

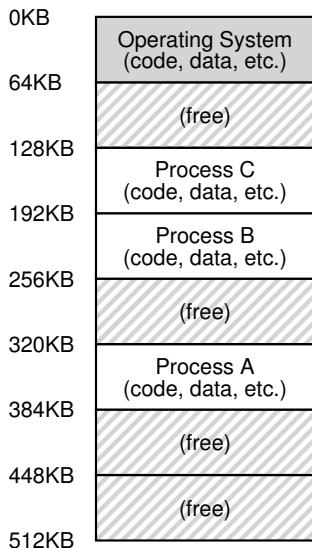


Figure 13.2: Three Processes: Sharing Memory

13.3 The Address Space

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

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

In the example in Figure 13.3 (page 4), we have a tiny address space (only 16KB)¹. The program code lives at the top of the address space

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

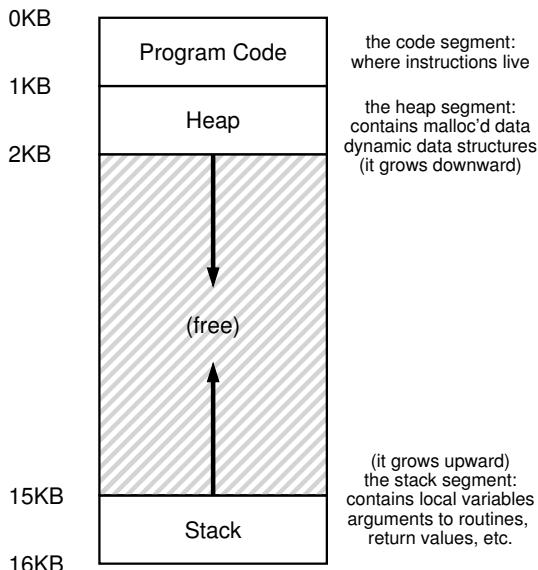


Figure 13.3: An Example Address Space

(starting at 0 in this example, and is packed into the first 1K of the address space). Code is static (and thus easy to place in memory), so we can place it at the top of the address space and know that it won't need any more space as the program runs.

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

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

THE CRUX: HOW TO VIRTUALIZE MEMORY

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

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

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

13.4 Goals

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

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

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

Finally, a third VM goal is **protection**. The OS should make sure to **protect** processes from one another as well as the OS itself from pro-

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

TIP: THE PRINCIPLE OF ISOLATION

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

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

In the next chapters, we'll focus our exploration on the basic **mechanisms** needed to virtualize memory, including hardware and operating systems support. We'll also investigate some of the more relevant **policies** that you'll encounter in operating systems, including how to manage free space and which pages to kick out of memory when you run low on space. In doing so, we'll build up your understanding of how a modern virtual memory system really works³.

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!

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

ASIDE: EVERY ADDRESS YOU SEE IS VIRTUAL

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

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

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(int argc, char *argv[]) {
4     printf("location of code : %p\n", main);
5     printf("location of heap : %p\n", malloc(100e6));
6     int x = 3;
7     printf("location of stack: %p\n", &x);
8     return x;
9 }
```

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

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

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

References

- [BH70] "The Nucleus of a Multiprogramming System" by Per Brinch Hansen. Communications of the ACM, 13:4, April 1970. *The first paper to suggest that the OS, or kernel, should be a minimal and flexible substrate for building customized operating systems; this theme is revisited throughout OS research history.*
- [CV65] "Introduction and Overview of the Multics System" by F. J. Corbato, V. A. Vyssotsky. Fall Joint Computer Conference, 1965. *A great early Multics paper. Here is the great quote about time sharing: "The impetus for time-sharing first arose from professional programmers because of their constant frustration in debugging programs at batch processing installations. Thus, the original goal was to time-share computers to allow simultaneous access by several persons while giving to each of them the illusion of having the whole machine at his disposal."*
- [DV66] "Programming Semantics for Multiprogrammed Computations" by Jack B. Dennis, Earl C. Van Horn. Communications of the ACM, Volume 9, Number 3, March 1966. *An early paper (but not the first) on multiprogramming.*
- [L60] "Man-Computer Symbiosis" by J. C. R. Licklider. IRE Transactions on Human Factors in Electronics, HFE-1:1, March 1960. *A funky paper about how computers and people are going to enter into a symbiotic age; clearly well ahead of its time but a fascinating read nonetheless.*
- [M62] "Time-Sharing Computer Systems" by J. McCarthy. Management and the Computer of the Future, MIT Press, Cambridge, MA, 1962. *Probably McCarthy's earliest recorded paper on time sharing. In another paper [M83], he claims to have been thinking of the idea since 1957. McCarthy left the systems area and went on to become a giant in Artificial Intelligence at Stanford, including the creation of the LISP programming language. See McCarthy's home page for more info: <http://www-formal.stanford.edu/jmc/>*
- [M+63] "A Time-Sharing Debugging System for a Small Computer" by J. McCarthy, S. Boilen, E. Fredkin, J. C. R. Licklider. AFIPS '63 (Spring), New York, NY, May 1963. *A great early example of a system that swapped program memory to the "drum" when the program wasn't running, and then back into "core" memory when it was about to be run.*
- [M83] "Reminiscences on the History of Time Sharing" by John McCarthy. 1983. Available: <http://www-formal.stanford.edu/jmc/history/timesharing/timesharing.html>. *A terrific historical note on where the idea of time-sharing might have come from, including some doubts towards those who cite Strachey's work [S59] as the by pioneering work in this area..*
- [NS07] "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation" by N. Nethercote, J. Seward. PLDI 2007, San Diego, California, June 2007. *Valgrind is a lifesaver of a program for those who use unsafe languages like C. Read this paper to learn about its very cool binary instrumentation techniques – it's really quite impressive.*
- [R+89] "Mach: A System Software kernel" by R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, M. Jones. COMPON '89, February 1989. *Although not the first project on microkernels per se, the Mach project at CMU was well-known and influential; it still lives today deep in the bowels of Mac OS X.*
- [S59] "Time Sharing in Large Fast Computers" by C. Strachey. Proceedings of the International Conference on Information Processing, UNESCO, June 1959. *One of the earliest references on time sharing.*
- [S+03] "Improving the Reliability of Commodity Operating Systems" by M. M. Swift, B. N. Bershad, H. M. Levy. SOSP '03. *The first paper to show how microkernel-like thinking can improve operating system reliability.*

Homework (Code)

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

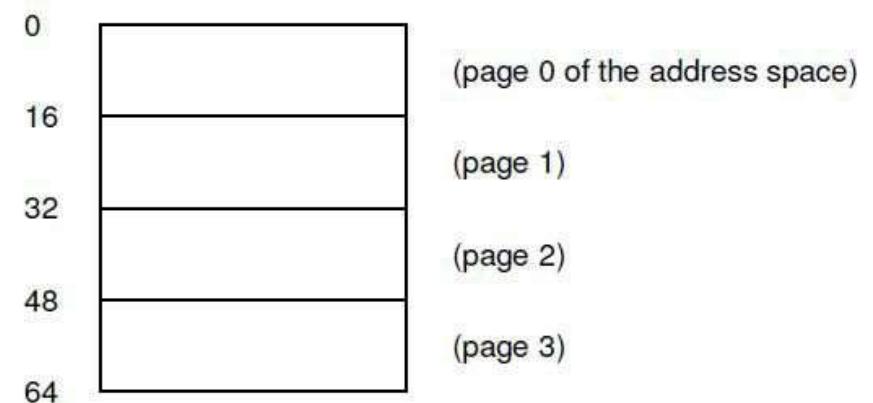
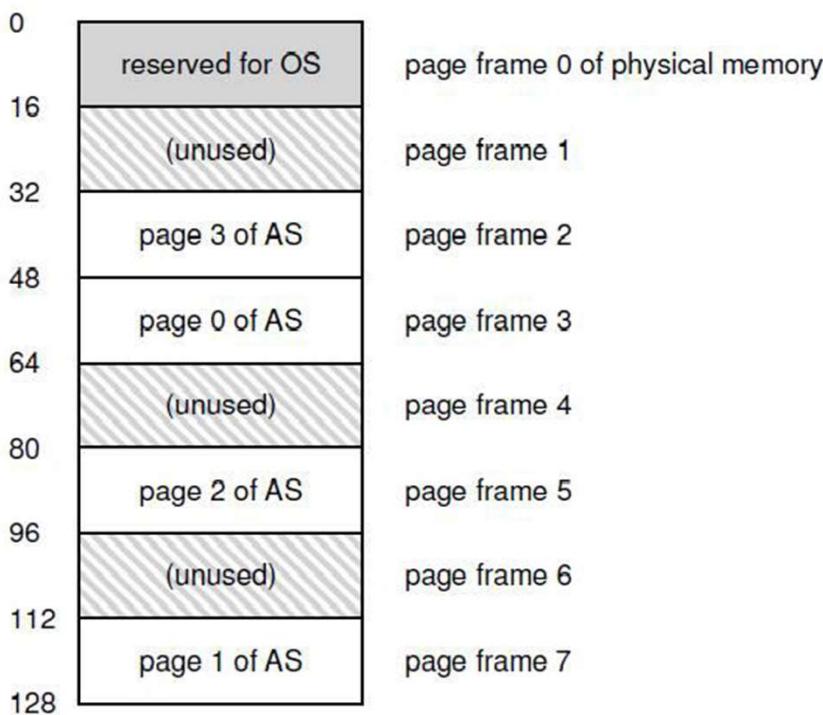
Questions

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

Paging

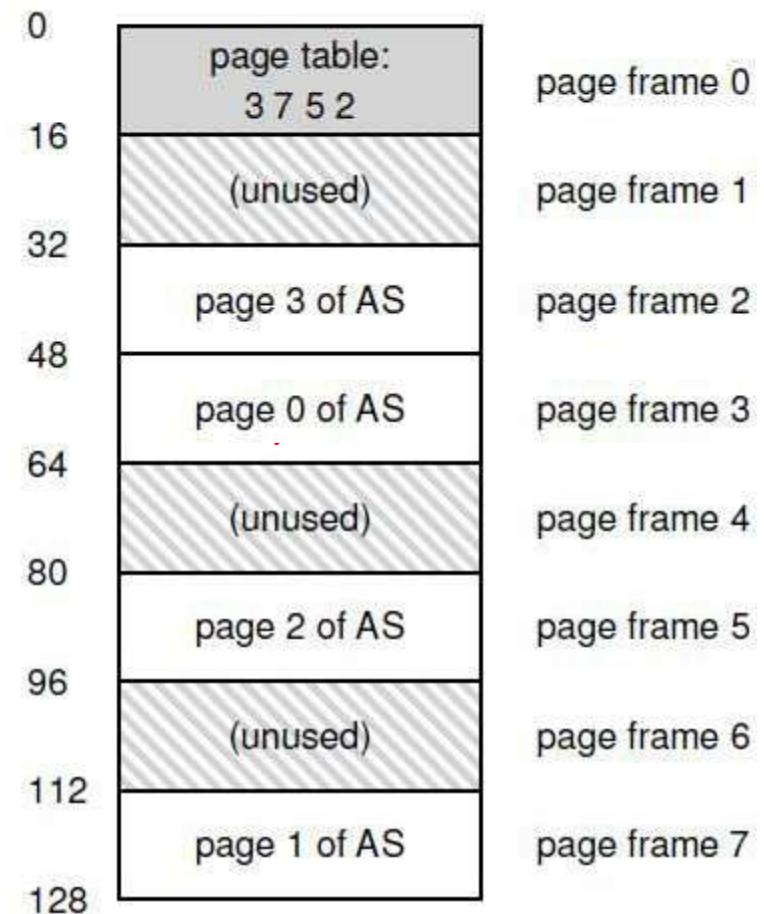
Paging

- Allocate memory in fixed size chunks (“pages”)
- Avoids external fragmentation (no small “holes”)
- Has internal fragmentation (partially filled pages)



Page table

- Per process data structure to help VA-PA translation
- Array stores mappings from virtual page number (VPN) to physical frame number (PFN)
 - E.g., VP 0 → PF 3, VP 1 → PF 7
- Part of OS memory (in PCB)
- MMU has access to page table and uses it for address translation
- OS updates page table upon context switch

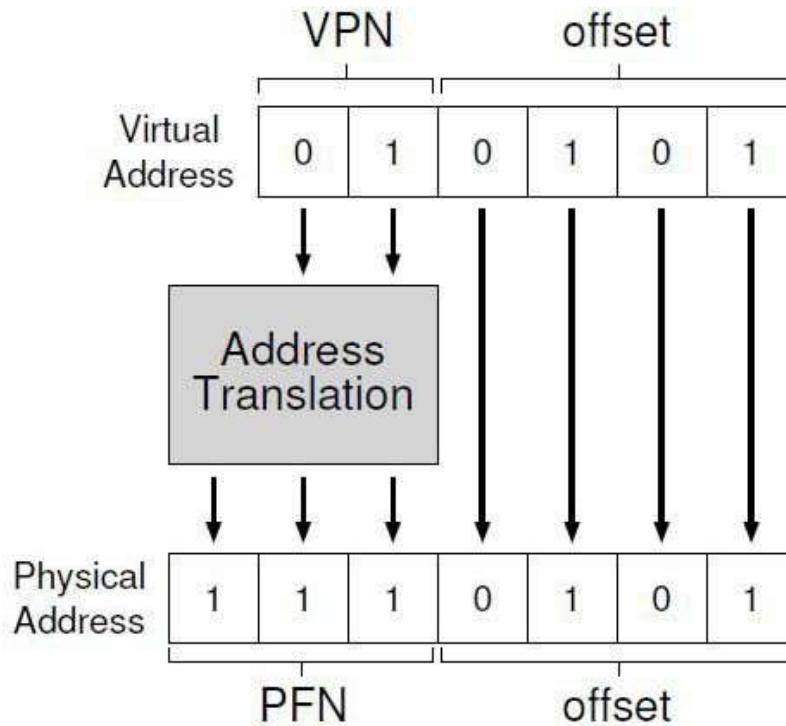


Page table entry (PTE)

- Simplest page table: linear page table
- Page table is an array of page table entries, one per virtual page
- VPN (virtual page no.) is index into this array
- Each PTE contains PFN (physical frame number) and few other bits
 - Valid bit: is this page used by process?
 - Protection bits: read/write permissions
 - Present bit: is this page in memory? (more later)
 - Dirty bit: has this page been modified?
 - Accessed bit: has this page been recently accessed?

Address translation in hardware

- Most significant bits of VA give the VPN
- Page table maps VPN to PFN
- PA is obtained from PFN and offset within a page
- MMU stores (physical) address of start of page table, not all entries.
- “Walks” the page table to get relevant PTE



What happens on memory access?

- CPU requests code or data at a virtual address
- MMU must translate VA to PA
 - First, access memory to read page table entry
 - Translate VA to PA
 - Then, access memory to fetch code/data
- Paging adds overhead to memory access
- Solution? A cache for VA-PA mappings

Translation Lookaside Buffer (TLB)

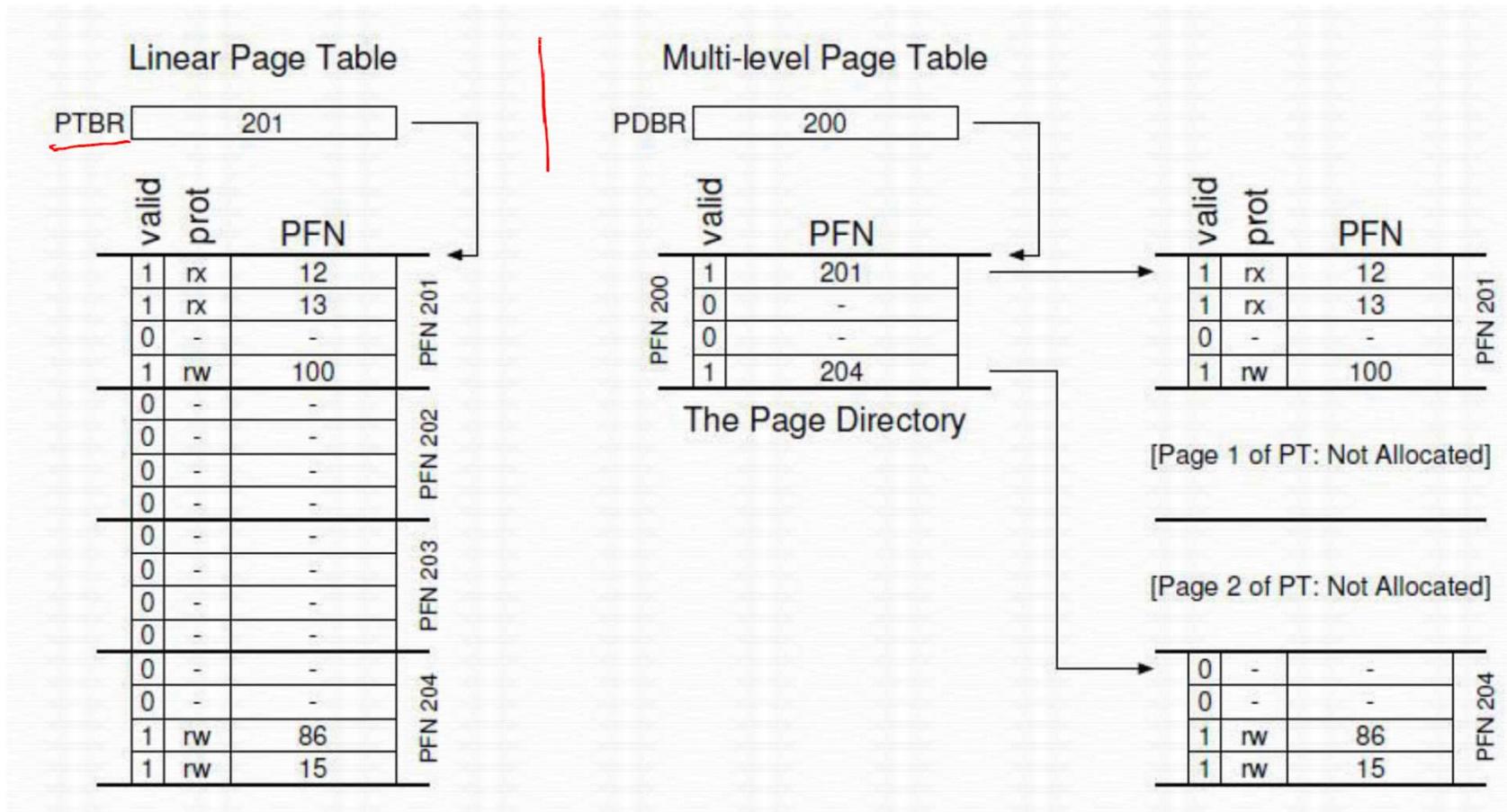
- A cache of recent VA-PA mappings
- To translate VA to PA, MMU first looks up TLB
- If TLB hit, PA can be directly used
- If TLB miss, then MMU performs additional memory accesses to “walk” page table
- TLB misses are expensive (multiple memory accesses)
 - Locality of reference helps to have high hit rate
- TLB entries may become invalid on context switch and change of page tables

How are page tables stored in memory?

- What is typical size of page table?
 - 32 bit VA, 4 KB pages, so $2^{32} / 2^{12} = 2^{20}$ entries
 - If each PTE is 4 bytes, then page table is 4MB
 - One such page table per process!
- How to reduce the size of page tables?
 - Larger pages, so fewer entries
- How does OS allocate memory for such large tables?
 - Page table is itself split into smaller chunks!

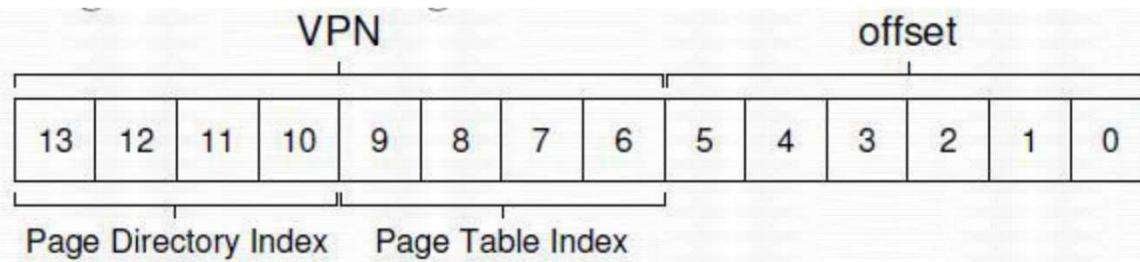
Multilevel page tables (1)

- A page table is spread over many pages
- An “outer” page table or page directory tracks the PFNs of the page table pages



Multilevel page tables (2)

- Depending on how large the page table is, we may need more than 2 levels also
 - 64-bit architectures may need 7 levels
- What about address translation?
 - First few bits of VA to identify outer page table entry
 - Next few bits to index into next level of PTEs



- In case of TLB miss, multiple accesses to memory required to access all the levels of page tables

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 bits¹; in the book, we'll often use tiny examples to make them easier to digest.

¹A 64-bit address space is hard to imagine, it is so amazingly large. An analogy might help: if you think of a 32-bit address space as the size of a tennis court, a 64-bit address space is about the size of Europe(!).

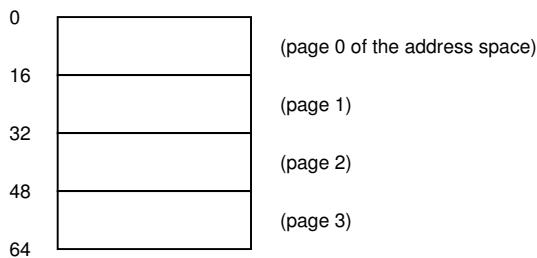


Figure 18.1: A Simple 64-byte Address Space

Physical memory, as shown in Figure 18.2, also consists of a number of fixed-sized slots, in this case eight page frames (making for a 128-byte physical memory, also ridiculously small). As you can see in the diagram, the pages of the virtual address space have been placed at different locations throughout physical memory; the diagram also shows the OS using some of physical memory for itself.

Paging, as we will see, has a number of advantages over our previous approaches. Probably the most important improvement will be *flexibility*: with a fully-developed paging approach, the system will be able to support the abstraction of an address space effectively, regardless of how a process uses the address space; we won't, for example, make assumptions about the direction the heap and stack grow and how they are used.

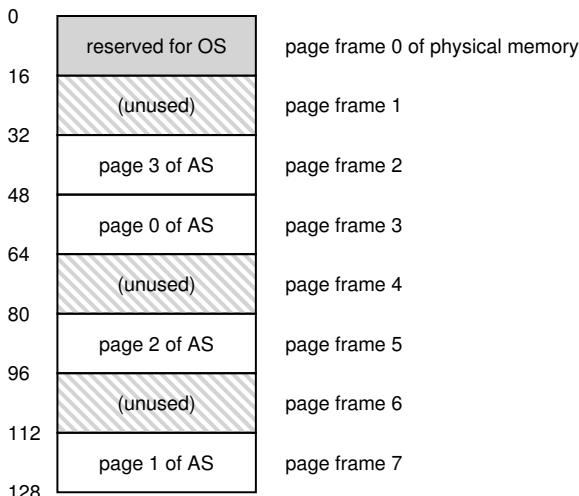


Figure 18.2: A 64-Byte Address Space In A 128-Byte Physical Memory

Another advantage is the *simplicity* of free-space management that paging affords. For example, when the OS wishes to place our tiny 64-byte address space into our eight-page physical memory, it simply finds four free pages; perhaps the OS keeps a **free list** of all free pages for this, and just grabs the first four free pages off of this list. In the example, the OS has placed virtual page 0 of the address space (AS) in physical frame 3, virtual page 1 of the AS in physical frame 7, page 2 in frame 5, and page 3 in frame 2. Page frames 1, 4, and 6 are currently free.

To record where each virtual page of the address space is placed in physical memory, the operating system usually keeps a *per-process* data structure known as a **page table**. The major role of the page table is to store **address translations** for each of the virtual pages of the address space, thus letting us know where in physical memory each page resides. For our simple example (Figure 18.2, page 2), the page table would thus have the following four entries: (Virtual Page 0 → Physical Frame 3), (VP 1 → PF 7), (VP 2 → PF 5), and (VP 3 → PF 2).

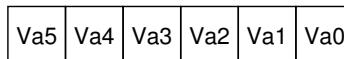
It is important to remember that this page table is a *per-process* data structure (most page table structures we discuss are per-process structures; an exception we'll touch on is the **inverted page table**). If another process were to run in our example above, the OS would have to manage a different 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:

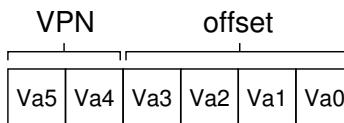
```
movl <virtual address>, %eax
```

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:

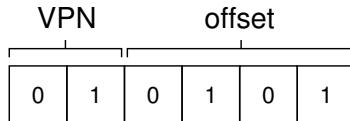


The page size is 16 bytes in a 64-byte address space; thus we need to be able to select 4 pages, and the top 2 bits of the address do just that. Thus, we have a 2-bit virtual page number (VPN). The remaining bits tell us which byte of the page we are interested in, 4 bits in this case; we call this the 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).

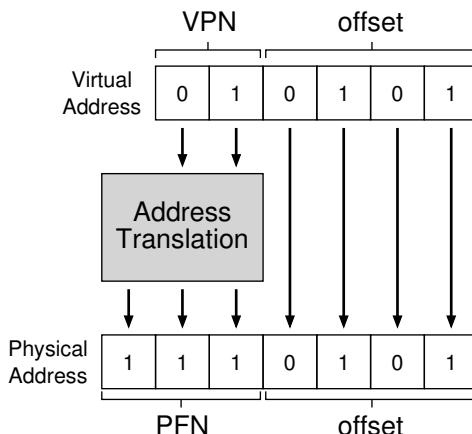


Figure 18.3: The Address Translation Process

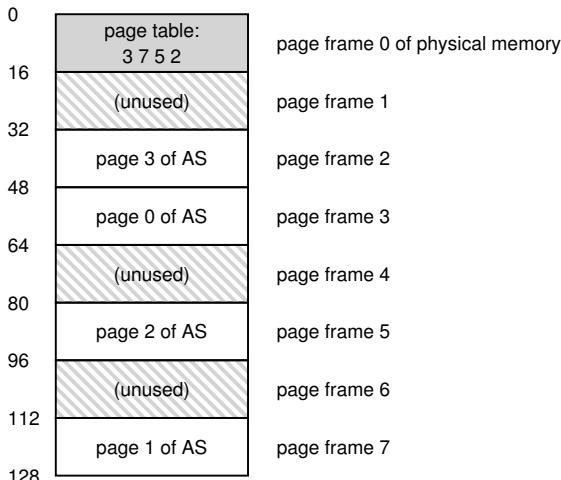


Figure 18.4: Example: Page Table in Kernel Physical Memory

Note the offset stays the same (i.e., it is not translated), because the offset just tells us which byte *within* the page we want. Our final physical address is 1110101 (117 in decimal), and is exactly where we want our load to fetch data from (Figure 18.2, page 2).

With this basic overview in mind, we can now ask (and hopefully, answer) a few basic questions you may have about paging. For example, where are these page tables stored? What are the typical contents of the page table, and how big are the tables? Does paging make the system (too) slow? These and other beguiling questions are answered, at least in part, in the text below. Read on!

18.2 Where Are Page Tables Stored?

Page tables can get terribly large, much bigger than the small segment table or base/bounds pair we have discussed previously. For example, imagine a typical 32-bit address space, with 4KB pages. This virtual address splits into a 20-bit VPN and 12-bit offset (recall that 10 bits would be needed for a 1KB page size, and just add two more to get to 4KB).

A 20-bit VPN implies that there are 2^{20} translations that the OS would have to manage for each process (that's roughly a million); assuming we need 4 bytes per **page table entry (PTE)** to hold the physical translation plus any other useful stuff, we get an immense 4MB of memory needed for each page table! That is pretty large. Now imagine there are 100 processes running: this means the OS would need 400MB of memory just for all those address translations! Even in the modern era, where

ASIDE: DATA STRUCTURE — THE PAGE TABLE

One of the most important data structures in the memory management subsystem of a modern OS is the **page table**. In general, a page table stores **virtual-to-physical address translations**, thus letting the system know where each page of an address space actually resides in physical memory. Because each address space requires such translations, in general there is one page table per process in the system. The exact structure of the page table is either determined by the hardware (older systems) or can be more flexibly managed by the OS (modern systems).

machines have gigabytes of memory, it seems a little crazy to use a large chunk of it just for translations, no? And we won't even think about how big such a page table would be for a 64-bit address space; that would be too gruesome and perhaps scare you off entirely.

Because page tables are so big, we don't keep any special on-chip hardware in the MMU to store the page table of the currently-running process. Instead, we store the page table for each process in *memory* somewhere. Let's assume for now that the page tables live in physical memory that the OS manages; later we'll see that much of OS memory itself can be virtualized, and thus page tables can be stored in OS virtual memory (and even swapped to disk), but that is too confusing right now, so we'll ignore it. In Figure 18.4 (page 5) is a picture of a page table in OS memory; see the tiny set of translations in there?

18.3 What's Actually In The Page Table?

Let's talk a little about page table organization. The page table is just a data structure that is used to map virtual addresses (or really, virtual page numbers) to physical addresses (physical frame numbers). Thus, any data structure could work. The simplest form is called a **linear page table**, which is just an array. The OS *indexes* the array by the virtual page number (VPN), and looks up the page-table entry (PTE) at that index in order to find the desired physical frame number (PFN). For now, we will assume this simple linear structure; in later chapters, we will make use of more advanced data structures to help solve some problems with paging.

As for the contents of each PTE, we have a number of different bits in there worth understanding at some level. A **valid bit** is common to indicate whether the particular translation is valid; for example, when a program starts running, it will have code and heap at one end of its address space, and the stack at the other. All the unused space in-between will be marked **invalid**, and if the process tries to access such memory, it will generate a trap to the OS which will likely terminate the process. Thus, the valid bit is crucial for supporting a sparse address space; by simply marking all the unused pages in the address space invalid, we remove the need to allocate physical frames for those pages and thus save a great deal of memory.

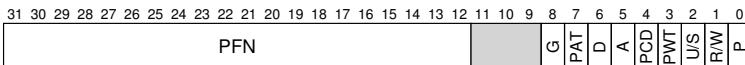


Figure 18.5: An x86 Page Table Entry (PTE)

We also might have **protection bits**, indicating whether the page could be read from, written to, or executed from. Again, accessing a page in a way not allowed by these bits will generate a trap to the OS.

There are a couple of other bits that are important but we won't talk about much for now. A **present bit** indicates whether this page is in physical memory or on disk (i.e., it has been **swapped out**). We will understand this machinery further when we study how to **swap** parts of the address space to disk to support address spaces that are larger than physical memory; swapping allows the OS to free up physical memory by moving rarely-used pages to disk. A **dirty bit** is also common, indicating whether the page has been modified since it was brought into memory.

A **reference bit** (a.k.a. **accessed bit**) is sometimes used to track whether a page has been accessed, and is useful in determining which pages are popular and thus should be kept in memory; such knowledge is critical during **page replacement**, a topic we will study in great detail in subsequent chapters.

Figure 18.5 shows an example page table entry from the x86 architecture [I09]. It contains a present bit (P); a read/write bit (R/W) which determines if writes are allowed to this page; a user/supervisor bit (U/S) which determines if user-mode processes can access the page; a few bits (PWT, PCD, PAT, and G) that determine how hardware caching works for these pages; an accessed bit (A) and a dirty bit (D); and finally, the page frame number (PFN) itself.

Read the Intel Architecture Manuals [I09] for more details on x86 paging support. Be forewarned, however; reading manuals such as these, while quite informative (and certainly necessary for those who write code to use such page tables in the OS), can be challenging at first. A little patience, and a lot of desire, is required.

ASIDE: WHY NO VALID BIT?

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

18.4 Paging: Also Too Slow

With page tables in memory, we already know that they might be too big. As it turns out, they can slow things down too. For example, take our simple instruction:

```
movl 21, %eax
```

Again, let's just examine the explicit reference to address 21 and not worry about the instruction fetch. In this example, we'll assume the hardware performs the translation for us. To fetch the desired data, the system must first **translate** the virtual address (21) into the correct physical address (117). Thus, before fetching the data from address 117, the system must first fetch the proper page table entry from the process's page table, perform the translation, and then load the data from physical memory.

To do so, the hardware must know where the page table is for the currently-running process. Let's assume for now that a single **page-table base register** contains the physical address of the starting location of the page table. To find the location of the desired PTE, the hardware will thus perform the following functions:

```
VPN      = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))
```

In our example, `VPN_MASK` would be set to 0x30 (hex 30, or binary 110000) which picks out the VPN bits from the full virtual address; `SHIFT` is set to 4 (the number of bits in the offset), such that we move the VPN bits down to form the correct integer virtual page number. For example, with virtual address 21 (010101), and masking turns this value into 010000; the shift turns it into 01, or virtual page 1, as desired. We then use this value as an index into the array of PTEs pointed to by the page table base register.

Once this physical address is known, the hardware can fetch the PTE from memory, extract the PFN, and concatenate it with the offset from the virtual address to form the desired physical address. Specifically, you can think of the PFN being left-shifted by `SHIFT`, and then bitwise OR'd with the offset to form the final address as follows:

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

Finally, the hardware can fetch the desired data from memory and put it into register `eax`. The program has now succeeded at loading a value from memory!

To summarize, we now describe the initial protocol for what happens on each memory reference. Figure 18.6 (page 9) shows the approach. For every memory reference (whether an instruction fetch or an explicit load or store), paging requires us to perform one extra memory reference in order to first fetch the translation from the page table. That is a lot of

```

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(SEGMENTATIONFAULT)
13 else if (CanAccess(PTE.ProtectBits) == False)
14     RaiseException(PROTECTIONFAULT)
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

work! Extra memory references are costly, and in this case will likely slow down the process by a factor of two or more.

And now you can hopefully see that there are *two* real problems that we must solve. Without careful design of both hardware and software, page tables will cause the system to run too slowly, as well as take up too much memory. While seemingly a great solution for our memory virtualization needs, these two crucial problems must first be overcome.

18.5 A Memory Trace

Before closing, we now trace through a simple memory access example to demonstrate all of the resulting memory accesses that occur when using paging. The code snippet (in C, in a file called `array.c`) that we are interested in is as follows:

```

int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;

```

We compile `array.c` and run it with the following commands:

```

prompt> gcc -o array array.c -Wall -O
prompt> ./array

```

Of course, to truly understand what memory accesses this code snippet (which simply initializes an array) will make, we'll have to know (or assume) a few more things. First, we'll have to **disassemble** the resulting binary (using `objdump` on Linux, or `otool` on a Mac) to see what assembly instructions are used to initialize the array in a loop. Here is the resulting assembly code:

```
1024 movl $0x0, (%edi,%eax,4)
1028 incl %eax
1032 cmpl $0x03e8,%eax
1036 jne 0x1024
```

The code, if you know a little **x86**, is actually quite easy to understand². The first instruction moves the value zero (shown as `$0x0`) into the virtual memory address of the location of the array; this address is computed by taking the contents of `%edi` and adding `%eax` multiplied by four to it. Thus, `%edi` holds the base address of the array, whereas `%eax` holds the array index (*i*); we multiply by four because the array is an array of integers, each of size four bytes.

The second instruction increments the array index held in `%eax`, and the third instruction compares the contents of that register to the hex value `0x03e8`, or decimal 1000. If the comparison shows that two values are not yet equal (which is what the `jne` instruction tests), the fourth instruction jumps back to the top of the loop.

To understand which memory accesses this instruction sequence makes (at both the virtual and physical levels), we'll have to assume something about where in virtual memory the code snippet and array are found, as well as the contents and location of the page table.

For this example, we assume a virtual address space of size 64KB (unrealistically small). We also assume a page size of 1KB.

All we need to know now are the contents of the page table, and its location in physical memory. Let's assume we have a linear (array-based) page table and that it is located at physical address 1KB (1024).

As for its contents, there are just a few virtual pages we need to worry about having mapped for this example. First, there is the virtual page the code lives on. Because the page size is 1KB, virtual address 1024 resides on the second page of the virtual address space (VPN=1, as VPN=0 is the first page). Let's assume this virtual page maps to physical frame 4 (VPN 1 → 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).

²We are cheating a little bit here, assuming each instruction is four bytes in size for simplicity; in actuality, x86 instructions are variable-sized.

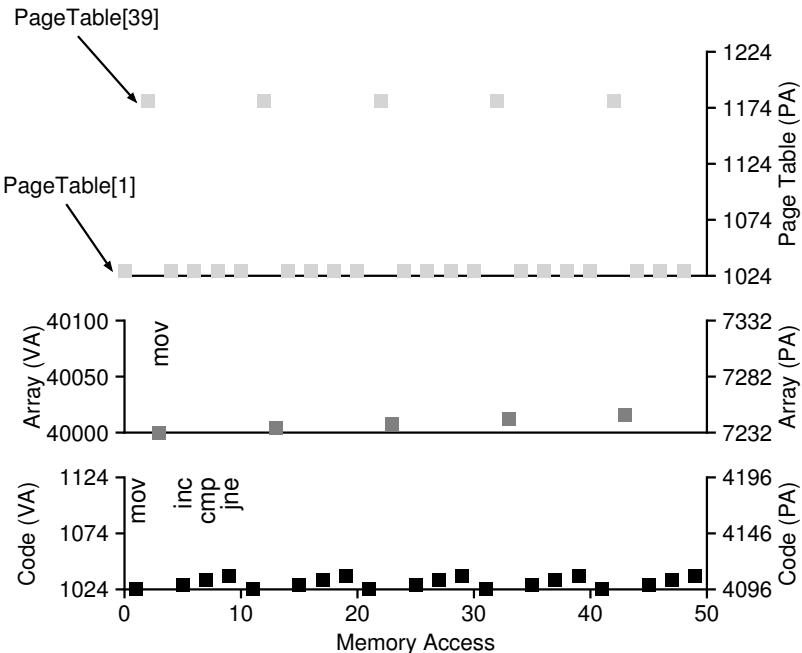


Figure 18.7: A Virtual (And Physical) Memory Trace

We are now ready to trace the memory references of the program. When it runs, each instruction fetch will generate two memory references: one to the page table to find the physical frame that the instruction resides within, and one to the instruction itself to fetch it to the CPU for processing. In addition, there is one explicit memory reference in the form of the `mov` instruction; this adds another page table access first (to translate the array virtual address to the correct physical one) and then the array access itself.

The entire process, for the first five loop iterations, is depicted in Figure 18.7 (page 11). The bottom most graph shows the instruction memory references on the y-axis in black (with virtual addresses on the left, and the actual physical addresses on the right); the middle graph shows array accesses in dark gray (again with virtual on left and physical on right); finally, the topmost graph shows page table memory accesses in light gray (just physical, as the page table in this example resides in physical memory). The x-axis, for the entire trace, shows memory accesses across the first five iterations of the loop; there are 10 memory accesses per loop, which includes four instruction fetches, one explicit update of memory, and five page table accesses to translate those four fetches and one explicit update.

See if you can make sense of the patterns that show up in this visualization. In particular, what will change as the loop continues to run beyond these first five iterations? Which new memory locations will be accessed? Can you figure it out?

This has just been the simplest of examples (only a few lines of C code), and yet you might already be able to sense the complexity of understanding the actual memory behavior of real applications. Don't worry: it definitely gets worse, because the mechanisms we are about to introduce only complicate this already complex machinery. Sorry³!

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.

³We're not really sorry. But, we are sorry about not being sorry, if that makes sense.

References

[KE+62] “One-level Storage System” by T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner. IRE Trans. EC-11, 2, 1962. Reprinted in Bell and Newell, “Computer Structures: Readings and Examples”. McGraw-Hill, New York, 1971. *The Atlas pioneered the idea of dividing memory into fixed-sized pages and in many senses was an early form of the memory-management ideas we see in modern computer systems.*

[I09] “Intel 64 and IA-32 Architectures Software Developer’s Manuals” Intel, 2009. Available: <http://www.intel.com/products/processor/manuals>. In particular, pay attention to “Volume 3A: System Programming Guide Part 1” and “Volume 3B: System Programming Guide Part 2”.

[L78] “The Manchester Mark I and Atlas: A Historical Perspective” by S. H. Lavington. Communications of the ACM, Volume 21:1, January 1978. *This paper is a great retrospective of some of the history of the development of some important computer systems. As we sometimes forget in the US, many of these new ideas came from overseas.*

Homework (Simulation)

In this homework, you will use a simple program, which is known as `paging-linear-translate.py`, to see if you understand how simple virtual-to-physical address translation works with linear page tables. See the README for details.

Questions

1. Before doing any translations, let's use the simulator to study how linear page tables change size given different parameters. Compute the size of linear page tables as different parameters change. Some suggested inputs are below; by using the `-v` flag, you can see how many page-table entries are filled. First, to understand how linear page table size changes as the address space grows, run with these flags:

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

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

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

Before running any of these, try to think about the expected trends. How should page-table size change as the address space grows? As the page size grows? Why not use big pages in general?

2. Now let's do some translations. Start with some small examples, and change the number of pages that are allocated to the address space with the `-u` flag. For example:

```
-P 1k -a 16k -p 32k -v -u 0  
-P 1k -a 16k -p 32k -v -u 25  
-P 1k -a 16k -p 32k -v -u 50  
-P 1k -a 16k -p 32k -v -u 75  
-P 1k -a 16k -p 32k -v -u 100
```

What happens as you increase the percentage of pages that are allocated in each address space?

3. Now let's try some different random seeds, and some different (and sometimes quite crazy) address-space parameters, for variety:

```
-P 8 -a 32 -p 1024 -v -s 1  
-P 8k -a 32k -p 1m -v -s 2  
-P 1m -a 256m -p 512m -v -s 3
```

Which of these parameter combinations are unrealistic? Why?

4. Use the program to try out some other problems. Can you find the limits of where the program doesn't work anymore? For example, what happens if the address-space size is *bigger* than physical memory?

Demand Paging

Is main memory always enough?

- Are all pages of all active processes always in main memory?
 - Not necessary, with large address spaces
- OS uses a part of disk (swap space) to store pages that are not in active use

	PFN 0	PFN 1	PFN 2	PFN 3			
Physical Memory	Proc 0 [VPN 0]	Proc 1 [VPN 2]	Proc 1 [VPN 3]	Proc 2 [VPN 0]			
	Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6
Swap Space	Proc 0 [VPN 1]	Proc 0 [VPN 2]	[Free]	Proc 1 [VPN 0]	Proc 1 [VPN 1]	Proc 3 [VPN 0]	Proc 2 [VPN 1]

Page fault

- Present bit in page table entry: indicates if a page of a process resides in memory or not
- When translating VA to PA, MMU reads present bit
- If page present in memory, directly accessed
- If page not in memory, MMU raises a trap to the OS – page fault

Page fault handling

- Page fault traps OS and moves CPU to kernel mode
- OS fetches disk address of page and issues read to disk
 - OS keeps track of disk address (say, in page table)
 - OS context switches to another process
 - Current process is blocked and cannot run
- When disk read completes, OS updates page table of process, and marks it as ready
- When process scheduled again, OS restarts the instruction that caused page fault

Summary: what happens on memory access

- CPU issues load to a VA for code or data
 - Checks CPU cache first
 - Goes to main memory in case of cache miss
- MMU looks up TLB for VA
 - If TLB hit, obtains PA, fetches memory location and returns to CPU (via CPU caches)
 - If TLB miss, MMU accesses memory, walks page table, and obtains page table entry
 - If present bit set in PTE, accesses memory
 - If not present but valid, raises page fault. OS handles page fault and restarts the CPU load instruction
 - If invalid page access, trap to OS for illegal access

More complications in a page fault

- When servicing page fault, what if OS finds that there is no free page to swap in the faulting page?
- OS must swap out an existing page (if it has been modified, i.e., dirty) and then swap in the faulting page – too much work!
- OS may proactively swap out pages to keep list of free pages handy
- Which pages to swap out? Decided by page replacement policy.

Page replacement policies

- Optimal: replace page not needed for longest time in future (not practical!)
- FIFO: replace page that was brought into memory earliest (may be a popular page!)
- LRU/LFU: replace the page that was least recently (or frequently) used in the past

Example: Optimal policy

- Example: 3 frames for 4 pages (0,1,2,3)
- First few accesses are cold (compulsory) misses

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

Example: FIFO

- Usually worse than optimal
- Belady's anomaly: performance may get worse when memory size increases!

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

Example: LRU

- Equivalent to optimal in this simple example
- Works well due to locality of references

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

How is LRU implemented?

- OS is not involved in every memory access – how does it know which page is LRU?
- Hardware help and some approximations
- MMU sets a bit in PTE (“accessed” bit) when a page is accessed
- OS periodically looks at this bit to estimate pages that are active and inactive
- To replace, OS tries to find a page that does not have access bit set
 - May also look for page with dirty bit not set (to avoid swapping out to disk)

Paging: Faster Translations (TLBs)

Using paging as the core mechanism to support virtual memory can lead to high performance overheads. By chopping the address space into small, fixed-sized units (i.e., pages), paging requires a large amount of mapping information. Because that mapping information is generally stored in physical memory, paging logically requires an extra memory lookup for each virtual address generated by the program. Going to memory for translation information before every instruction fetch or explicit load or store is prohibitively slow. And thus our problem:

THE CRUX:
HOW TO SPEED UP ADDRESS TRANSLATION

How can we speed up address translation, and generally avoid the extra memory reference that paging seems to require? What hardware support is required? What OS involvement is needed?

When we want to make things fast, the OS usually needs some help. And help often comes from the OS's old friend: the hardware. To speed address translation, we are going to add what is called (for historical reasons [CP78]) a **translation-lookaside buffer**, or **TLB** [CG68, C95]. A TLB is part of the chip's **memory-management unit (MMU)**, and is simply a hardware **cache** of popular virtual-to-physical address translations; thus, a better name would be an **address-translation cache**. Upon each virtual memory reference, the hardware first checks the TLB to see if the desired translation is held therein; if so, the translation is performed (quickly) *without* having to consult the page table (which has all translations). Because of their tremendous performance impact, TLBs in a real sense make virtual memory possible [C95].

```

1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True) // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10 else // TLB Miss
11     PTEAddr = PTBR + (VPN * sizeof(PTE))
12     PTE = AccessMemory(PTEAddr)
13     if (PTE.Valid == False)
14         RaiseException(SEGMENTATION_FAULT)
15     else if (CanAccess(PTE.ProtectBits) == False)
16         RaiseException(PROTECTION_FAULT)
17     else
18         TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19     RetryInstruction()

```

Figure 19.1: **TLB Control Flow Algorithm**

19.1 TLB Basic Algorithm

Figure 19.1 shows a rough sketch of how hardware might handle a virtual address translation, assuming a simple **linear page table** (i.e., the page table is an array) and a **hardware-managed TLB** (i.e., the hardware handles much of the responsibility of page table accesses; we'll explain more about this below).

The algorithm the hardware follows works like this: first, extract the virtual page number (VPN) from the virtual address (Line 1 in Figure 19.1), and check if the TLB holds the translation for this VPN (Line 2). If it does, we have a **TLB hit**, which means the TLB holds the translation. Success! We can now extract the page frame number (PFN) from the relevant TLB entry, concatenate that onto the offset from the original virtual address, and form the desired physical address (PA), and access memory (Lines 5–7), assuming protection checks do not fail (Line 4).

If the CPU does not find the translation in the TLB (a **TLB miss**), we have some more work to do. In this example, the hardware accesses the page table to find the translation (Lines 11–12), and, assuming that the virtual memory reference generated by the process is valid and accessible (Lines 13, 15), updates the TLB with the translation (Line 18). These set of actions are costly, primarily because of the extra memory reference needed to access the page table (Line 12). Finally, once the TLB is updated, the hardware retries the instruction; this time, the translation is found in the TLB, and the memory reference is processed quickly.

The TLB, like all caches, is built on the premise that in the common case, translations are found in the cache (i.e., are hits). If so, little overhead is added, as the TLB is found near the processing core and is designed to be quite fast. When a miss occurs, the high cost of paging is incurred; the page table must be accessed to find the translation, and an extra memory reference (or more, with more complex page tables) results. If this happens often, the program will likely run noticeably more slowly; memory accesses, relative to most CPU instructions, are quite costly, and TLB misses lead to more memory accesses. Thus, it is our hope to avoid TLB misses as much as we can.

19.2 Example: Accessing An Array

To make clear the operation of a TLB, let's examine a simple virtual address trace and see how a TLB can improve its performance. In this example, let's assume we have an array of 10 4-byte integers in memory, starting at virtual address 100. Assume further that we have a small 8-bit virtual address space, with 16-byte pages; thus, a virtual address breaks down into a 4-bit VPN (there are 16 virtual pages) and a 4-bit offset (there are 16 bytes on each of those pages).

Figure 19.2 (page 4) shows the array laid out on the 16 16-byte pages of the system. As you can see, the array's first entry ($a[0]$) begins on ($VPN=06$, $offset=04$); only three 4-byte integers fit onto that page. The array continues onto the next page ($VPN=07$), where the next four entries ($a[3] \dots a[6]$) 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

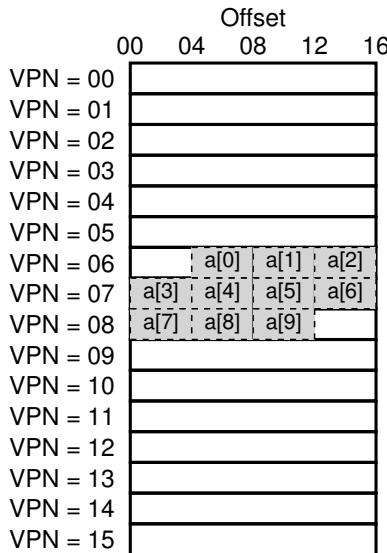


Figure 19.2: Example: An Array In A Tiny Address Space

into the TLB. And hence the reason for our success. Access to $a[2]$ encounters similar success (another hit), because it too lives on the same page as $a[0]$ and $a[1]$.

Unfortunately, when the program accesses $a[3]$, we encounter another TLB miss. However, once again, the next entries ($a[4] \dots a[6]$) 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

TIP: USE CACHING WHEN POSSIBLE

Caching is one of the most fundamental performance techniques in computer systems, one that is used again and again to make the “common-case fast” [HP06]. The idea behind hardware caches is to take advantage of **locality** in instruction and data references. There are usually two types of locality: **temporal locality** and **spatial locality**. With temporal locality, the idea is that an instruction or data item that has been recently accessed will likely be re-accessed soon in the future. Think of loop variables or instructions in a loop; they are accessed repeatedly over time. With spatial locality, the idea is that if a program accesses memory at address x , it will likely soon access memory near x . Imagine here streaming through an array of some kind, accessing one element and then the next. Of course, these properties depend on the exact nature of the program, and thus are not hard-and-fast laws but more like rules of thumb.

Hardware caches, whether for instructions, data, or address translations (as in our TLB) take advantage of locality by keeping copies of memory in small, fast on-chip memory. Instead of having to go to a (slow) memory to satisfy a request, the processor can first check if a nearby copy exists in a cache; if it does, the processor can access it quickly (i.e., in a few CPU cycles) and avoid spending the costly time it takes to access memory (many nanoseconds).

You might be wondering: if caches (like the TLB) are so great, why don’t we just make bigger caches and keep all of our data in them? Unfortunately, this is where we run into more fundamental laws like those of physics. If you want a fast cache, it has to be small, as issues like the speed-of-light and other physical constraints become relevant. Any large cache by definition is slow, and thus defeats the purpose. Thus, we are stuck with small, fast caches; the question that remains is how to best use them to improve performance.

had simply been twice as big (32 bytes, not 16), the array access would suffer even fewer misses. As typical page sizes are more like 4KB, these types of dense, array-based accesses achieve excellent TLB performance, encountering only a single miss per page of accesses.

One last point about TLB performance: if the program, soon after this loop completes, accesses the array again, we’d likely see an even better result, assuming that we have a big enough TLB to cache the needed translations: hit, hit, hit, hit, hit, hit, hit, hit, hit, hit. In this case, the TLB hit rate would be high because of **temporal locality**, i.e., the quick re-referencing of memory items in **time**. Like any cache, TLBs rely upon both spatial and temporal locality for success, which are program properties. If the program of interest exhibits such locality (and many programs do), the TLB hit rate will likely be high.

```

1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True) // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10     else // TLB Miss
11         RaiseException(TLB_MISS)

```

Figure 19.3: TLB Control Flow Algorithm (OS Handled)

19.3 Who Handles The TLB Miss?

One question that we must answer: who handles a TLB miss? Two answers are possible: the hardware, or the software (OS). In the olden days, the hardware had complex instruction sets (sometimes called **CISC**, for complex-instruction set computers) and the people who built the hardware didn't much trust those sneaky OS people. Thus, the hardware would handle the TLB miss entirely. To do this, the hardware has to know exactly *where* the page tables are located in memory (via a **page-table base register**, used in Line 11 in Figure 19.1), as well as their *exact format*; on a miss, the hardware would "walk" the page table, find the correct page-table entry and extract the desired translation, update the TLB with the translation, and retry the instruction. An example of an "older" architecture that has **hardware-managed TLBs** is the Intel x86 architecture, which uses a fixed **multi-level page table** (see the next chapter for details); the current page table is pointed to by the CR3 register [109].

More modern architectures (e.g., MIPS R10k [H93] or Sun's SPARC v9 [WG00], both **RISC** or reduced-instruction set computers) have what is known as a **software-managed TLB**. On a TLB miss, the hardware simply raises an exception (line 11 in Figure 19.3), which pauses the current instruction stream, raises the privilege level to kernel mode, and jumps to a **trap handler**. As you might guess, this trap handler is code within the OS that is written with the express purpose of handling TLB misses. When run, the code will lookup the translation in the page table, use special "privileged" instructions to update the TLB, and return from the trap; at this point, the hardware retries the instruction (resulting in a TLB hit).

Let's discuss a couple of important details. First, the return-from-trap instruction needs to be a little different than the return-from-trap we saw before when servicing a system call. In the latter case, the return-from-trap should resume execution at the instruction *after* the trap into the OS, just as a return from a procedure call returns to the instruction immediately following the call into the procedure. In the former case, when returning from a TLB miss-handling trap, the hardware must resume execution at the instruction that *caused* the trap; this retry thus lets the in-

ASIDE: RISC vs. CISC

In the 1980's, a great battle took place in the computer architecture community. On one side was the **CISC** camp, which stood for **Complex Instruction Set Computing**; on the other side was **RISC**, for **Reduced Instruction Set Computing** [PS81]. The RISC side was spear-headed by David Patterson at Berkeley and John Hennessy at Stanford (who are also co-authors of some famous books [HP06]), although later John Cocke was recognized with a Turing award for his earliest work on RISC [CM00]. CISC instruction sets tend to have a lot of instructions in them, and each instruction is relatively powerful. For example, you might see a string copy, which takes two pointers and a length and copies bytes from source to destination. The idea behind CISC was that instructions should be high-level primitives, to make the assembly language itself easier to use, and to make code more compact.

RISC instruction sets are exactly the opposite. A key observation behind RISC is that instruction sets are really compiler targets, and all compilers really want are a few simple primitives that they can use to generate high-performance code. Thus, RISC proponents argued, let's rip out as much from the hardware as possible (especially the microcode), and make what's left simple, uniform, and fast.

In the early days, RISC chips made a huge impact, as they were noticeably faster [BC91]; many papers were written; a few companies were formed (e.g., MIPS and Sun). However, as time progressed, CISC manufacturers such as Intel incorporated many RISC techniques into the core of their processors, for example by adding early pipeline stages that transformed complex instructions into micro-instructions which could then be processed in a RISC-like manner. These innovations, plus a growing number of transistors on each chip, allowed CISC to remain competitive. The end result is that the debate died down, and today both types of processors can be made to run fast.

struction run again, this time resulting in a TLB hit. Thus, depending on how a trap or exception was caused, the hardware must save a different PC when trapping into the OS, in order to resume properly when the time to do so arrives.

Second, when running the TLB miss-handling code, the OS needs to be extra careful not to cause an infinite chain of TLB misses to occur. Many solutions exist; for example, you could keep TLB miss handlers in physical memory (where they are **unmapped** and not subject to address translation), or reserve some entries in the TLB for permanently-valid translations and use some of those permanent translation slots for the handler code itself; these **wired** translations always hit in the TLB.

The primary advantage of the software-managed approach is *flexibility*: the OS can use any data structure it wants to implement the page

ASIDE: TLB VALID BIT \neq PAGE TABLE VALID BIT

A common mistake is to confuse the valid bits found in a TLB with those found in a page table. In a page table, when a page-table entry (PTE) is marked invalid, it means that the page has not been allocated by the process, and should not be accessed by a correctly-working program. The usual response when an invalid page is accessed is to trap to the OS, which will respond by killing the process.

A TLB valid bit, in contrast, simply refers to whether a TLB entry has a valid translation within it. When a system boots, for example, a common initial state for each TLB entry is to be set to invalid, because no address translations are yet cached there. Once virtual memory is enabled, and once programs start running and accessing their virtual address spaces, the TLB is slowly populated, and thus valid entries soon fill the TLB.

The TLB valid bit is quite useful when performing a context switch too, as we'll discuss further below. By setting all TLB entries to invalid, the system can ensure that the about-to-be-run process does not accidentally use a virtual-to-physical translation from a previous process.

table, without necessitating hardware change. Another advantage is *simplicity*, as seen in the TLB control flow (line 11 in Figure 19.3, in contrast to lines 11–19 in Figure 19.1). The hardware doesn't do much on a miss: just raise an exception and let the OS TLB miss handler do the rest.

19.4 TLB Contents: What's In There?

Let's look at the contents of the hardware TLB in more detail. A typical TLB might have 32, 64, or 128 entries and be what is called **fully associative**. Basically, this just means that any given translation can be anywhere in the TLB, and that the hardware will search the entire TLB in parallel to find the desired translation. A TLB entry might look like this:

VPN | PFN | other bits

Note that both the VPN and PFN are present in each entry, as a translation could end up in any of these locations (in hardware terms, the TLB is known as a **fully-associative cache**). The hardware searches the entries in parallel to see if there is a match.

More interesting are the "other bits". For example, the TLB commonly has a **valid** bit, which says whether the entry has a valid translation or not. Also common are **protection** bits, which determine how a page can be accessed (as in the page table). For example, code pages might be marked *read and execute*, whereas heap pages might be marked *read and write*. There may also be a few other fields, including an **address-space identifier**, a **dirty bit**, and so forth; see below for more information.

19.5 TLB Issue: Context Switches

With TLBs, some new issues arise when switching between processes (and hence address spaces). Specifically, the TLB contains virtual-to-physical translations that are only valid for the currently running process; these translations are not meaningful for other processes. As a result, when switching from one process to another, the hardware or OS (or both) must be careful to ensure that the about-to-be-run process does not accidentally use translations from some previously run process.

To understand this situation better, let's look at an example. When one process (P1) is running, it assumes the TLB might be caching translations that are valid for it, i.e., that come from P1's page table. Assume, for this example, that the 10th virtual page of P1 is mapped to physical frame 100.

In this example, assume another process (P2) exists, and the OS soon might decide to perform a context switch and run it. Assume here that the 10th virtual page of P2 is mapped to physical frame 170. If entries for both processes were in the TLB, the contents of the TLB would be:

VPN	PFN	valid	prot
10	100	1	rwx
—	—	0	—
10	170	1	rwx
—	—	0	—

In the TLB above, we clearly have a problem: VPN 10 translates to either PFN 100 (P1) or PFN 170 (P2), but the hardware can't distinguish which entry is meant for which process. Thus, we need to do some more work in order for the TLB to correctly and efficiently support virtualization across multiple processes. And thus, a crux:

THE CRUX:

HOW TO MANAGE TLB CONTENTS ON A CONTEXT SWITCH

When context-switching between processes, the translations in the TLB for the last process are not meaningful to the about-to-be-run process. What should the hardware or OS do in order to solve this problem?

There are a number of possible solutions to this problem. One approach is to simply **flush** the TLB on context switches, thus emptying it before running the next process. On a software-based system, this can be accomplished with an explicit (and privileged) hardware instruction; with a hardware-managed TLB, the flush could be enacted when the page-table base register is changed (note the OS must change the PTBR on a context switch anyhow). In either case, the flush operation simply sets all valid bits to 0, essentially clearing the contents of the TLB.

By flushing the TLB on each context switch, we now have a working solution, as a process will never accidentally encounter the wrong trans-

lations in the TLB. However, there is a cost: each time a process runs, it must incur TLB misses as it touches its data and code pages. If the OS switches between processes frequently, this cost may be high.

To reduce this overhead, some systems add hardware support to enable sharing of the TLB across context switches. In particular, some hardware systems provide an **address space identifier (ASID)** field in the TLB. You can think of the ASID as a **process identifier (PID)**, but usually it has fewer bits (e.g., 8 bits for the ASID versus 32 bits for a PID).

If we take our example TLB from above and add ASIDs, it is clear processes can readily share the TLB: only the ASID field is needed to differentiate otherwise identical translations. Here is a depiction of a TLB with the added ASID field:

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

Thus, with address-space identifiers, the TLB can hold translations from different processes at the same time without any confusion. Of course, the hardware also needs to know which process is currently running in order to perform translations, and thus the OS must, on a context switch, set some privileged register to the ASID of the current process.

As an aside, you may also have thought of another case where two entries of the TLB are remarkably similar. In this example, there are two entries for two different processes with two different VPNs that point to the *same* physical page:

VPN	PFN	valid	prot	ASID
10	101	1	r-x	1
—	—	0	—	—
50	101	1	r-x	2
—	—	0	—	—

This situation might arise, for example, when two processes *share* a page (a code page, for example). In the example above, Process 1 is sharing physical page 101 with Process 2; P1 maps this page into the 10th page of its address space, whereas P2 maps it to the 50th page of its address space. Sharing of code pages (in binaries, or shared libraries) is useful as it reduces the number of physical pages in use, thus reducing memory overheads.

19.6 Issue: Replacement Policy

As with any cache, and thus also with the TLB, one more issue that we must consider is **cache replacement**. Specifically, when we are installing a new entry in the TLB, we have to **replace** an old one, and thus the question: which one to replace?

THE CRUX: HOW TO DESIGN TLB REPLACEMENT POLICY

Which TLB entry should be replaced when we add a new TLB entry? The goal, of course, being to minimize the **miss rate** (or increase **hit rate**) and thus improve performance.

We will study such policies in some detail when we tackle the problem of swapping pages to disk; here we'll just highlight a few typical policies. One common approach is to evict the **least-recently-used** or **LRU** entry. LRU tries to take advantage of locality in the memory-reference stream, assuming it is likely that an entry that has not recently been used is a good candidate for eviction. Another typical approach is to use a **random** policy, which evicts a TLB mapping at random. Such a policy is useful due to its simplicity and ability to avoid corner-case behaviors; for example, a "reasonable" policy such as LRU behaves quite unreasonably when a program loops over $n + 1$ pages with a TLB of size n ; in this case, LRU misses upon every access, whereas random does much better.

19.7 A Real TLB Entry

Finally, let's briefly look at a real TLB. This example is from the MIPS R4000 [H93], a modern system that uses software-managed TLBs; a slightly simplified MIPS TLB entry can be seen in Figure 19.4.

The MIPS R4000 supports a 32-bit address space with 4KB pages. Thus, we would expect a 20-bit VPN and 12-bit offset in our typical virtual address. However, as you can see in the TLB, there are only 19 bits for the VPN; as it turns out, user addresses will only come from half the address space (the rest reserved for the kernel) and hence only 19 bits of VPN are needed. The VPN translates to up to a 24-bit physical frame number (PFN), and hence can support systems with up to 64GB of (physical) main memory (2^{24} 4KB pages).

There are a few other interesting bits in the MIPS TLB. We see a *global* bit (G), which is used for pages that are globally-shared among processes. Thus, if the global bit is set, the ASID is ignored. We also see the 8-bit *ASID*, which the OS can use to distinguish between address spaces (as

0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	3	3		
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	
VPN										G	ASID										PFN									
C										D	V																			

Figure 19.4: A MIPS TLB Entry

TIP: RAM ISN'T ALWAYS RAM (CULLER'S LAW)

The term **random-access memory**, or **RAM**, implies that you can access any part of RAM just as quickly as another. While it is generally good to think of RAM in this way, because of hardware/OS features such as the TLB, accessing a particular page of memory may be costly, particularly if that page isn't currently mapped by your TLB. Thus, it is always good to remember the implementation tip: **RAM isn't always RAM**. Sometimes randomly accessing your address space, particularly if the number of pages accessed exceeds the TLB coverage, can lead to severe performance penalties. Because one of our advisors, David Culler, used to always point to the TLB as the source of many performance problems, we name this law in his honor: **Culler's Law**.

described above). One question for you: what should the OS do if there are more than 256 (2^8) processes running at a time? Finally, we see 3 *Coherence* (C) bits, which determine how a page is cached by the hardware (a bit beyond the scope of these notes); a *dirty* bit which is marked when the page has been written to (we'll see the use of this later); a *valid* bit which tells the hardware if there is a valid translation present in the entry. There is also a *page mask* field (not shown), which supports multiple page sizes; we'll see later why having larger pages might be useful. Finally, some of the 64 bits are unused (shaded gray in the diagram).

MIPS TLBs usually have 32 or 64 of these entries, most of which are used by user processes as they run. However, a few are reserved for the OS. A *wired* register can be set by the OS to tell the hardware how many slots of the TLB to reserve for the OS; the OS uses these reserved mappings for code and data that it wants to access during critical times, where a TLB miss would be problematic (e.g., in the TLB miss handler).

Because the MIPS TLB is software managed, there needs to be instructions to update the TLB. The MIPS provides four such instructions: `TLBP`, which probes the TLB to see if a particular translation is in there; `TLBR`, which reads the contents of a TLB entry into registers; `TLBWI`, which replaces a specific TLB entry; and `TLBWR`, which replaces a random TLB entry. The OS uses these instructions to manage the TLB's contents. It is of course critical that these instructions are **privileged**; imagine what a user process could do if it could modify the contents of the TLB (hint: just about anything, including take over the machine, run its own malicious "OS", or even make the Sun disappear).

19.8 Summary

We have seen how hardware can help us make address translation faster. By providing a small, dedicated on-chip TLB as an address-translation cache, most memory references will hopefully be handled *without* having to access the page table in main memory. Thus, in the common case,

the performance of the program will be almost as if memory isn't being virtualized at all, an excellent achievement for an operating system, and certainly essential to the use of paging in modern systems.

However, TLBs do not make the world rosy for every program that exists. In particular, if the number of pages a program accesses in a short period of time exceeds the number of pages that fit into the TLB, the program will generate a large number of TLB misses, and thus run quite a bit more slowly. We refer to this phenomenon as exceeding the **TLB coverage**, and it can be quite a problem for certain programs. One solution, as we'll discuss in the next chapter, is to include support for larger page sizes; by mapping key data structures into regions of the program's address space that are mapped by larger pages, the effective coverage of the TLB can be increased. Support for large pages is often exploited by programs such as a **database management system** (a **DBMS**), which have certain data structures that are both large and randomly-accessed.

One other TLB issue worth mentioning: TLB access can easily become a bottleneck in the CPU pipeline, in particular with what is called a **physically-indexed cache**. With such a cache, address translation has to take place *before* the cache is accessed, which can slow things down quite a bit. Because of this potential problem, people have looked into all sorts of clever ways to access caches with *virtual* addresses, thus avoiding the expensive step of translation in the case of a cache hit. Such a **virtually-indexed cache** solves some performance problems, but introduces new issues into hardware design as well. See Wiggins's fine survey for more details [W03].

References

- [BC91] "Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization" by D. Bhandarkar and Douglas W. Clark. Communications of the ACM, September 1991. *A great and fair comparison between RISC and CISC. The bottom line: on similar hardware, RISC was about a factor of three better in performance.*
- [CM00] "The evolution of RISC technology at IBM" by John Cocke, V. Markstein. IBM Journal of Research and Development, 44:1/2. *A summary of the ideas and work behind the IBM 801, which many consider the first true RISC microprocessor.*
- [C95] "The Core of the Black Canyon Computer Corporation" by John Couleur. IEEE Annals of History of Computing, 17:4, 1995. *In this fascinating historical note, Couleur talks about how he invented the TLB in 1964 while working for GE, and the fortuitous collaboration that thus ensued with the Project MAC folks at MIT.*
- [CG68] "Shared-access Data Processing System" by John F. Couleur, Edward L. Glaser. Patent 3412382, November 1968. *The patent that contains the idea for an associative memory to store address translations. The idea, according to Couleur, came in 1964.*
- [CP78] "The architecture of the IBM System/370" by R.P. Case, A. Padegs. Communications of the ACM, 21:1, 73-96, January 1978. *Perhaps the first paper to use the term translation lookaside buffer. The name arises from the historical name for a cache, which was a lookaside buffer as called by those developing the Atlas system at the University of Manchester; a cache of address translations thus became a translation lookaside buffer. Even though the term lookaside buffer fell out of favor, TLB seems to have stuck, for whatever reason.*
- [H93] "MIPS R4000 Microprocessor User's Manual". by Joe Heinrich. Prentice-Hall, June 1993. Available: http://cag.csail.mit.edu/raw/.documents/R4400_Uman_book_Ed2.pdf *A manual, one that is surprisingly readable. Or is it?*
- [HP06] "Computer Architecture: A Quantitative Approach" by John Hennessy and David Patterson. Morgan-Kaufmann, 2006. *A great book about computer architecture. We have a particular attachment to the classic first edition.*
- [I09] "Intel 64 and IA-32 Architectures Software Developer's Manuals" by Intel, 2009. Available: <http://www.intel.com/products/processor/manuals>. *In particular, pay attention to "Volume 3A: System Programming Guide" Part 1 and "Volume 3B: System Programming Guide Part 2".*
- [PS81] "RISC-I: A Reduced Instruction Set VLSI Computer" by D.A. Patterson and C.H. Sequin. ISCA '81, Minneapolis, May 1981. *The paper that introduced the term RISC, and started the avalanche of research into simplifying computer chips for performance.*
- [SB92] "CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking" by Rafael H. Saavedra-Barrera. EECS Department, University of California, Berkeley. Technical Report No. UCB/CSD-92-684, February 1992. *A great dissertation about how to predict execution time of applications by breaking them down into constituent pieces and knowing the cost of each piece. Probably the most interesting part that comes out of this work is the tool to measure details of the cache hierarchy (described in Chapter 5). Make sure to check out the wonderful diagrams therein.*
- [W03] "A Survey on the Interaction Between Caching, Translation and Protection" by Adam Wiggins. University of New South Wales TR UNSW-CSE-TR-0321, August, 2003. *An excellent survey of how TLBs interact with other parts of the CPU pipeline, namely hardware caches.*
- [WG00] "The SPARC Architecture Manual: Version 9" by David L. Weaver and Tom Germond. SPARC International, San Jose, California, September 2000. Available: www.sparc.org/standards/SPARCV9.pdf. *Another manual. I bet you were hoping for a more fun citation to end this chapter.*

Homework (Measurement)

In this homework, you are to measure the size and cost of accessing a TLB. The idea is based on work by Saavedra-Barrera [SB92], who developed a simple but beautiful method to measure numerous aspects of cache hierarchies, all with a very simple user-level program. Read his work for more details.

The basic idea is to access some number of pages within a large data structure (e.g., an array) and to time those accesses. For example, let's say the TLB size of a machine happens to be 4 (which would be very small, but useful for the purposes of this discussion). If you write a program that touches 4 or fewer pages, each access should be a TLB hit, and thus relatively fast. However, once you touch 5 pages or more, repeatedly in a loop, each access will suddenly jump in cost, to that of a TLB miss.

The basic code to loop through an array once should look like this:

```
int jump = PAGESIZE / sizeof(int);
for (i = 0; i < NUMPAGES * jump; i += jump)
    a[i] += 1;
```

In this loop, one integer per page of the array `a` is updated, up to the number of pages specified by `NUMPAGES`. By timing such a loop repeatedly (say, a few hundred million times in another loop around this one, or however many loops are needed to run for a few seconds), you can time how long each access takes (on average). By looking for jumps in cost as `NUMPAGES` increases, you can roughly determine how big the first-level TLB is, determine whether a second-level TLB exists (and how big it is if it does), and in general get a good sense of how TLB hits and misses can affect performance.

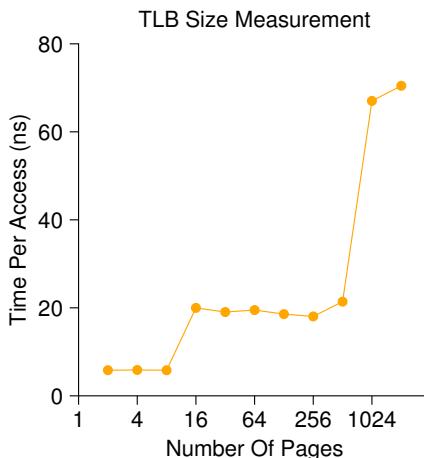


Figure 19.5: Discovering TLB Sizes and Miss Costs

Figure 19.5 (page 15) shows the average time per access as the number of pages accessed in the loop is increased. As you can see in the graph, when just a few pages are accessed (8 or fewer), the average access time is roughly 5 nanoseconds. When 16 or more pages are accessed, there is a sudden jump to about 20 nanoseconds per access. A final jump in cost occurs at around 1024 pages, at which point each access takes around 70 nanoseconds. From this data, we can conclude that there is a two-level TLB hierarchy; the first is quite small (probably holding between 8 and 16 entries); the second is larger but slower (holding roughly 512 entries). The overall difference between hits in the first-level TLB and misses is quite large, roughly a factor of fourteen. TLB performance matters!

Questions

1. For timing, you'll need to use a timer (e.g., `gettimeofday()`). How precise is such a timer? How long does an operation have to take in order for you to time it precisely? (this will help determine how many times, in a loop, you'll have to repeat a page access in order to time it successfully)
2. Write the program, called `tlb.c`, that can roughly measure the cost of accessing each page. Inputs to the program should be: the number of pages to touch and the number of trials.
3. Now write a script in your favorite scripting language (bash?) to run this program, while varying the number of pages accessed from 1 up to a few thousand, perhaps incrementing by a factor of two per iteration. Run the script on different machines and gather some data. How many trials are needed to get reliable measurements?
4. Next, graph the results, making a graph that looks similar to the one above. Use a good tool like `ploticus` or even `zplot`. Visualization usually makes the data much easier to digest; why do you think that is?
5. One thing to watch out for is compiler optimization. Compilers do all sorts of clever things, including removing loops which increment values that no other part of the program subsequently uses. How can you ensure the compiler does not remove the main loop above from your TLB size estimator?
6. Another thing to watch out for is the fact that most systems today ship with multiple CPUs, and each CPU, of course, has its own TLB hierarchy. To really get good measurements, you have to run your code on just one CPU, instead of letting the scheduler bounce it from one CPU to the next. How can you do that? (hint: look up “pinning a thread” on Google for some clues) What will happen if you don’t do this, and the code moves from one CPU to the other?
7. Another issue that might arise relates to initialization. If you don’t initialize the array `a` above before accessing it, the first time you access it will be very expensive, due to initial access costs such as demand zeroing. Will this affect your code and its timing? What can you do to counterbalance these potential costs?

Paging: Smaller Tables

We now tackle the second problem that paging introduces: page tables are too big and thus consume too much memory. Let's start out with a linear page table. As you might recall¹, linear page tables get pretty big. Assume again a 32-bit address space (2^{32} bytes), with 4KB (2^{12} byte) pages and a 4-byte page-table entry. An address space thus has roughly one million virtual pages in it ($\frac{2^{32}}{2^{12}}$); multiply by the page-table entry size and you see that our page table is 4MB in size. Recall also: we usually have one page table *for every process* in the system! With a hundred active processes (not uncommon on a modern system), we will be allocating hundreds of megabytes of memory just for page tables! As a result, we are in search of some techniques to reduce this heavy burden. There are a lot of them, so let's get going. But not before our crux:

CRUX: HOW TO MAKE PAGE TABLES SMALLER?

Simple array-based page tables (usually called linear page tables) are too big, taking up far too much memory on typical systems. How can we make page tables smaller? What are the key ideas? What inefficiencies arise as a result of these new data structures?

20.1 Simple Solution: Bigger Pages

We could reduce the size of the page table in one simple way: use bigger pages. Take our 32-bit address space again, but this time assume 16KB pages. We would thus have an 18-bit VPN plus a 14-bit offset. Assuming the same size for each PTE (4 bytes), we now have 2^{18} entries in our linear page table and thus a total size of 1MB per page table, a factor

¹Or indeed, you might not; this paging thing is getting out of control, no? That said, always make sure you understand the *problem* you are solving before moving onto the solution; indeed, if you understand the problem, you can often derive the solution yourself. Here, the problem should be clear: simple linear (array-based) page tables are too big.

ASIDE: MULTIPLE PAGE SIZES

As an aside, do note that many architectures (e.g., MIPS, SPARC, x86-64) now support multiple page sizes. Usually, a small (4KB or 8KB) page size is used. However, if a “smart” application requests it, a single large page (e.g., of size 4MB) can be used for a specific portion of the address space, enabling such applications to place a frequently-used (and large) data structure in such a space while consuming only a single TLB entry. This type of large page usage is common in database management systems and other high-end commercial applications. The main reason for multiple page sizes is not to save page table space, however; it is to reduce pressure on the TLB, enabling a program to access more of its address space without suffering from too many TLB misses. However, as researchers have shown [N+02], using multiple page sizes makes the OS virtual memory manager notably more complex, and thus large pages are sometimes most easily used simply by exporting a new interface to applications to request large pages directly.

of four reduction in size of the page table (not surprisingly, the reduction exactly mirrors the factor of four increase in page size).

The major problem with this approach, however, is that big pages lead to waste *within* each page, a problem known as **internal fragmentation** (as the waste is **internal** to the unit of allocation). Applications thus end up allocating pages but only using little bits and pieces of each, and memory quickly fills up with these overly-large pages. Thus, most systems use relatively small page sizes in the common case: 4KB (as in x86) or 8KB (as in SPARCv9). Our problem will not be solved so simply, alas.

20.2 Hybrid Approach: Paging and Segments

Whenever you have two reasonable but different approaches to something in life, you should always examine the combination of the two to see if you can obtain the best of both worlds. We call such a combination a **hybrid**. For example, why eat just chocolate or plain peanut butter when you can instead combine the two in a lovely hybrid known as the Reese’s Peanut Butter Cup [M28]?

Years ago, the creators of Multics (in particular Jack Dennis) chanced upon such an idea in the construction of the Multics virtual memory system [M07]. Specifically, Dennis had the idea of combining paging and segmentation in order to reduce the memory overhead of page tables. We can see why this might work by examining a typical linear page table in more detail. Assume we have an address space in which the used portions of the heap and stack are small. For the example, we use a tiny 16KB address space with 1KB pages (Figure 20.1); the page table for this address space is in Figure 20.2.

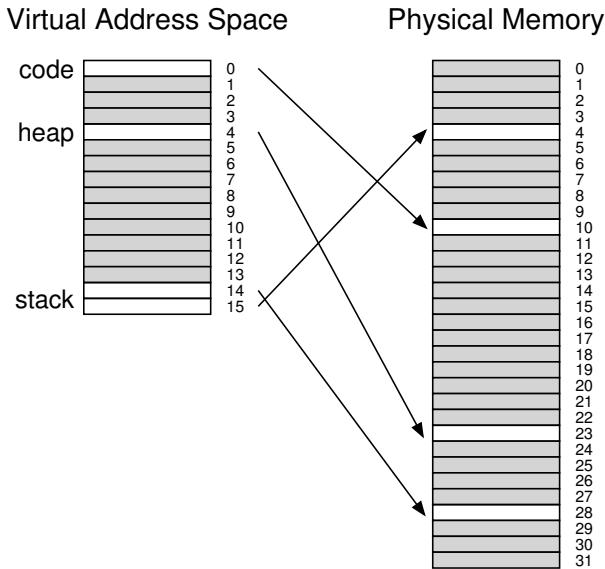


Figure 20.1: A 16KB Address Space With 1KB Pages

PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
23	1	rw-	1	1
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
28	1	rw-	1	1
4	1	rw-	1	1

Figure 20.2: A Page Table For 16KB Address Space

This example assumes the single code page (VPN 0) is mapped to physical page 10, the single heap page (VPN 4) to physical page 23, and the two stack pages at the other end of the address space (VPNs 14 and

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

Thus, our hybrid approach: instead of having a single page table for the entire address space of the process, why not have one per logical segment? In this example, we might thus have three page tables, one for the code, heap, and stack parts of the address space.

Now, remember with segmentation, we had a **base** register that told us where each segment lived in physical memory, and a **bound** or **limit** register that told us the size of said segment. In our hybrid, we still have those structures in the MMU; here, we use the base not to point to the segment itself but rather to hold the *physical address of the page table* of that segment. The bounds register is used to indicate the end of the page table (i.e., how many valid pages it has).

Let's do a simple example to clarify. Assume a 32-bit virtual address space with 4KB pages, and an address space split into four segments. We'll only use three segments for this example: one for code, one for heap, and one for stack.

To determine which segment an address refers to, we'll use the top two bits of the address space. Let's assume 00 is the unused segment, with 01 for code, 10 for the heap, and 11 for the stack. Thus, a virtual address looks like this:

3	3	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1
Seg	VPN																			Offset										

In the hardware, assume that there are thus three base/bounds pairs, one each for code, heap, and stack. When a process is running, the base register for each of these segments contains the physical address of a linear page table for that segment; thus, each process in the system now has *three* page tables associated with it. On a context switch, these registers must be changed to reflect the location of the page tables of the newly-running process.

On a TLB miss (assuming a hardware-managed TLB, i.e., where the hardware is responsible for handling TLB misses), the hardware uses the segment bits (SN) to determine which base and bounds pair to use. The hardware then takes the physical address therein and combines it with the VPN as follows to form the address of the page table entry (PTE):

```

SN           = (VirtualAddress & SEG_MASK) >> SN_SHIFT
VPN          = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))

```

This sequence should look familiar; it is virtually identical to what we saw before with linear page tables. The only difference, of course, is the use of one of three segment base registers instead of the single page table base register.

TIP: USE HYBRIDS

When you have two good and seemingly opposing ideas, you should always see if you can combine them into a **hybrid** that manages to achieve the best of both worlds. Hybrid corn species, for example, are known to be more robust than any naturally-occurring species. Of course, not all hybrids are a good idea; see the Zeedonk (or Zonkey), which is a cross of a Zebra and a Donkey. If you don't believe such a creature exists, look it up, and prepare to be amazed.

The critical difference in our hybrid scheme is the presence of a bounds register per segment; each bounds register holds the value of the maximum valid page in the segment. For example, if the code segment is using its first three pages (0, 1, and 2), the code segment page table will only have three entries allocated to it and the bounds register will be set to 3; memory accesses beyond the end of the segment will generate an exception and likely lead to the termination of the process. In this manner, our hybrid approach realizes a significant memory savings compared to the linear page table; unallocated pages between the stack and the heap no longer take up space in a page table (just to mark them as not valid).

However, as you might notice, this approach is not without problems. First, it still requires us to use segmentation; as we discussed before, segmentation is not quite as flexible as we would like, as it assumes a certain usage pattern of the address space; if we have a large but sparsely-used heap, for example, we can still end up with a lot of page table waste. Second, this hybrid causes external fragmentation to arise again. While most of memory is managed in page-sized units, page tables now can be of arbitrary size (in multiples of PTEs). Thus, finding free space for them in memory is more complicated. For these reasons, people continued to look for better ways to implement smaller page tables.

20.3 Multi-level Page Tables

A different approach doesn't rely on segmentation but attacks the same problem: how to get rid of all those invalid regions in the page table instead of keeping them all in memory? We call this approach a **multi-level page table**, as it turns the linear page table into something like a tree. This approach is so effective that many modern systems employ it (e.g., x86 [BOH10]). We now describe this approach in detail.

The basic idea behind a multi-level page table is simple. First, chop up the page table into page-sized units; then, if an entire page of page-table entries (PTEs) is invalid, don't allocate that page of the page table at all. To track whether a page of the page table is valid (and if valid, where it is in memory), use a new structure, called the **page directory**. The page directory thus either can be used to tell you where a page of the page table is, or that the entire page of the page table contains no valid pages.

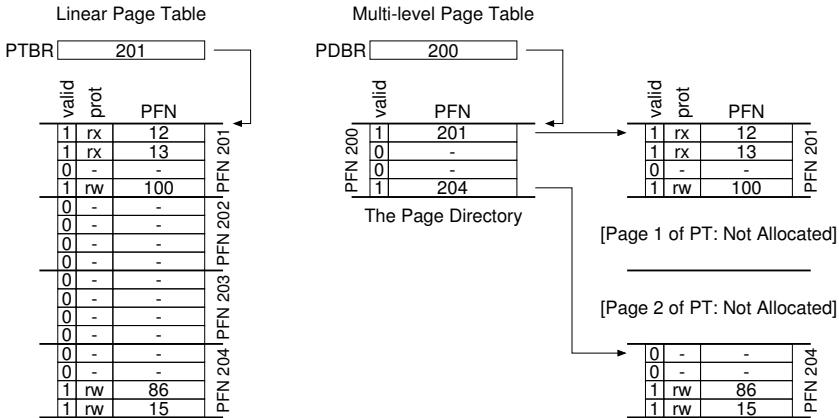


Figure 20.3: **Linear (Left) And Multi-Level (Right) Page Tables**

Figure 20.3 shows an example. On the left of the figure is the classic linear page table; even though most of the middle regions of the address space are not valid, we still require page-table space allocated for those regions (i.e., the middle two pages of the page table). On the right is a multi-level page table. The page directory marks just two pages of the page table as valid (the first and last); thus, just those two pages of the page table reside in memory. And thus you can see one way to visualize what a multi-level table is doing: it just makes parts of the linear page table disappear (freeing those frames for other uses), and tracks which pages of the page table are allocated with the page directory.

The page directory, in a simple two-level table, contains one entry per page of the page table. It consists of a number of **page directory entries (PDE)**. A PDE (minimally) has a **valid bit** and a **page frame number (PFN)**, similar to a PTE. However, as hinted at above, the meaning of this valid bit is slightly different: if the PDE is valid, it means that at least one of the pages of the page table that the entry points to (via the PFN) is valid, i.e., in at least one PTE on that page pointed to by this PDE, the valid bit in that PTE is set to one. If the PDE is not valid (i.e., equal to zero), the rest of the PDE is not defined.

Multi-level page tables have some obvious advantages over approaches we've seen thus far. First, and perhaps most obviously, the multi-level table only allocates page-table space in proportion to the amount of address space you are using; thus it is generally compact and supports sparse address spaces.

Second, if carefully constructed, each portion of the page table fits neatly within a page, making it easier to manage memory; the OS can simply grab the next free page when it needs to allocate or grow a page

TIP: UNDERSTAND TIME-SPACE TRADE-OFFS

When building a data structure, one should always consider **time-space trade-offs** in its construction. Usually, if you wish to make access to a particular data structure faster, you will have to pay a space-usage penalty for the structure.

table. Contrast this to a simple (non-paged) linear page table², which is just an array of PTEs indexed by VPN; with such a structure, the entire linear page table must reside contiguously in physical memory. For a large page table (say 4MB), finding such a large chunk of unused contiguous free physical memory can be quite a challenge. With a multi-level structure, we add a **level of indirection** through use of the page directory, which points to pieces of the page table; that indirection allows us to place page-table pages wherever we would like in physical memory.

It should be noted that there is a cost to multi-level tables; on a TLB miss, two loads from memory will be required to get the right translation information from the page table (one for the page directory, and one for the PTE itself), in contrast to just one load with a linear page table. Thus, the multi-level table is a small example of a **time-space trade-off**. We wanted smaller tables (and got them), but not for free; although in the common case (TLB hit), performance is obviously identical, a TLB miss suffers from a higher cost with this smaller table.

Another obvious negative is *complexity*. Whether it is the hardware or OS handling the page-table lookup (on a TLB miss), doing so is undoubtedly more involved than a simple linear page-table lookup. Often we are willing to increase complexity in order to improve performance or reduce overheads; in the case of a multi-level table, we make page-table lookups more complicated in order to save valuable memory.

A Detailed Multi-Level Example

To understand the idea behind multi-level page tables better, let's do an example. Imagine a small address space of size 16KB, with 64-byte pages. Thus, we have a 14-bit virtual address space, with 8 bits for the VPN and 6 bits for the offset. A linear page table would have 2^8 (256) entries, even if only a small portion of the address space is in use. Figure 20.4 (page 8) presents one example of such an address space.

In this example, virtual pages 0 and 1 are for code, virtual pages 4 and 5 for the heap, and virtual pages 254 and 255 for the stack; the rest of the pages of the address space are unused.

To build a two-level page table for this address space, we start with our full linear page table and break it up into page-sized units. Recall our full table (in this example) has 256 entries; assume each PTE is 4 bytes

²We are making some assumptions here, i.e., that all page tables reside in their entirety in physical memory (i.e., they are not swapped to disk); we'll soon relax this assumption.

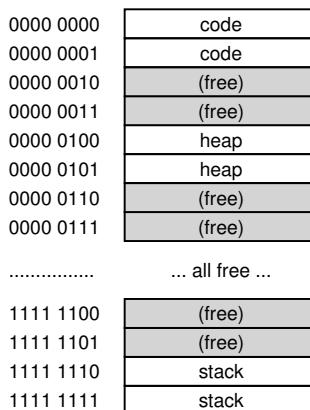
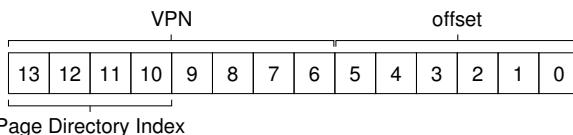


Figure 20.4: A 16KB Address Space With 64-byte Pages

in size. Thus, our page table is 1KB (256×4 bytes) in size. Given that we have 64-byte pages, the 1KB page table can be divided into 16 64-byte pages; each page can hold 16 PTEs.

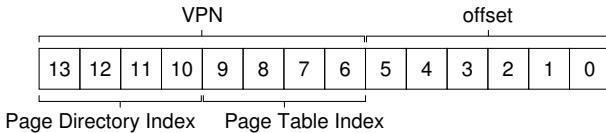
What we need to understand now is how to take a VPN and use it to index first into the page directory and then into the page of the page table. Remember that each is an array of entries; thus, all we need to figure out is how to construct the index for each from pieces of the VPN.

Let's first index into the page directory. Our page table in this example is small: 256 entries, spread across 16 pages. The page directory needs one entry per page of the page table; thus, it has 16 entries. As a result, we need four bits of the VPN to index into the directory; we use the top four bits of the VPN, as follows:



Once we extract the **page-directory index** (PDIndex for short) from the VPN, we can use it to find the address of the page-directory entry (PDE) with a simple calculation: $\text{PDEAddr} = \text{PageDirBase} + (\text{PDIndex} * \text{sizeof}(\text{PDE}))$. This results in our page directory, which we now examine to make further progress in our translation.

If the page-directory entry is marked invalid, we know that the access is invalid, and thus raise an exception. If, however, the PDE is valid, we have more work to do. Specifically, we now have to fetch the page-table entry (PTE) from the page of the page table pointed to by this page-directory entry. To find this PTE, we have to index into the portion of the page table using the remaining bits of the VPN:



This **page-table index** (PTIndex for short) can then be used to index into the page table itself, giving us the address of our PTE:

```
PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
```

Note that the page-frame number (PFN) obtained from the page-directory entry must be left-shifted into place before combining it with the page-table index to form the address of the PTE.

To see if this all makes sense, we'll now fill in a multi-level page table with some actual values, and translate a single virtual address. Let's begin with the **page directory** for this example (left side of Figure 20.5).

In the figure, you can see that each page directory entry (PDE) describes something about a page of the page table for the address space. In this example, we have two valid regions in the address space (at the beginning and end), and a number of invalid mappings in-between.

In physical page 100 (the physical frame number of the 0th page of the page table), we have the first page of 16 page table entries for the first 16 VPNs in the address space. See Figure 20.5 (middle part) for the contents of this portion of the page table.

This page of the page table contains the mappings for the first 16 VPNs; in our example, VPNs 0 and 1 are valid (the code segment), as

Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
101	1	—	0	—	55	1	rw-
				—	45	1	rw-

Figure 20.5: A Page Directory, And Pieces Of Page Table

TIP: BE WARY OF COMPLEXITY

System designers should be wary of adding complexity into their system. What a good systems builder does is implement the least complex system that achieves the task at hand. For example, if disk space is abundant, you shouldn't design a file system that works hard to use as few bytes as possible; similarly, if processors are fast, it is better to write a clean and understandable module within the OS than perhaps the most CPU-optimized, hand-assembled code for the task at hand. Be wary of needless complexity, in prematurely-optimized code or other forms; such approaches make systems harder to understand, maintain, and debug. As Antoine de Saint-Exupery famously wrote: "Perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away." What he didn't write: "It's a lot easier to say something about perfection than to actually achieve it."

are 4 and 5 (the heap). Thus, the table has mapping information for each of those pages. The rest of the entries are marked invalid.

The other valid page of the page table is found inside PFN 101. This page contains mappings for the last 16 VPNs of the address space; see Figure 20.5 (right) for details.

In the example, VPNs 254 and 255 (the stack) have valid mappings. Hopefully, what we can see from this example is how much space savings are possible with a multi-level indexed structure. In this example, instead of allocating the full *sixteen* pages for a linear page table, we allocate only *three*: one for the page directory, and two for the chunks of the page table that have valid mappings. The savings for large (32-bit or 64-bit) address spaces could obviously be much greater.

Finally, let's use this information in order to perform a translation. Here is an address that refers to the 0th byte of VPN 254: 0x3F80, or 11 1111 1000 0000 in binary.

Recall that we will use the top 4 bits of the VPN to index into the page directory. Thus, 1111 will choose the last (15th, if you start at the 0th) entry of the page directory above. This points us to a valid page of the page table located at address 101. We then use the next 4 bits of the VPN (1110) to index into that page of the page table and find the desired PTE. 1110 is the next-to-last (14th) entry on the page, and tells us that page 254 of our virtual address space is mapped at physical page 55. By concatenating PFN=55 (or hex 0x37) with offset=000000, we can thus form our desired physical address and issue the request to the memory system: $\text{PhysAddr} = (\text{PTE.PFN} \ll \text{SHIFT}) + \text{offset} = 00\ 1101\ 1100\ 0000 = 0x0DC0$.

You should now have some idea of how to construct a two-level page table, using a page directory which points to pages of the page table. Unfortunately, however, our work is not done. As we'll now discuss, sometimes two levels of page table is not enough!

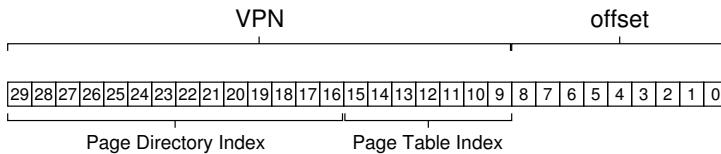
More Than Two Levels

In our example thus far, we've assumed that multi-level page tables only have two levels: a page directory and then pieces of the page table. In some cases, a deeper tree is possible (and indeed, needed).

Let's take a simple example and use it to show why a deeper multi-level table can be useful. In this example, assume we have a 30-bit virtual address space, and a small (512 byte) page. Thus our virtual address has a 21-bit virtual page number component and a 9-bit offset.

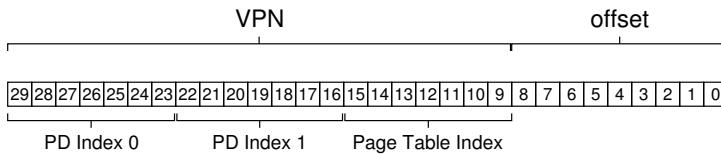
Remember our goal in constructing a multi-level page table: to make each piece of the page table fit within a single page. Thus far, we've only considered the page table itself; however, what if the page directory gets too big?

To determine how many levels are needed in a multi-level table to make all pieces of the page table fit within a page, we start by determining how many page-table entries fit within a page. Given our page size of 512 bytes, and assuming a PTE size of 4 bytes, you should see that you can fit 128 PTEs on a single page. When we index into a page of the page table, we can thus conclude we'll need the least significant 7 bits ($\log_2 128$) of the VPN as an index:



What you also might notice from the diagram above is how many bits are left into the (large) page directory: 14. If our page directory has 2^{14} entries, it spans not one page but 128, and thus our goal of making every piece of the multi-level page table fit into a page vanishes.

To remedy this problem, we build a further level of the tree, by splitting the page directory itself into multiple pages, and then adding another page directory on top of that, to point to the pages of the page directory. We can thus split up our virtual address as follows:



Now, when indexing the upper-level page directory, we use the very top bits of the virtual address (PD Index 0 in the diagram); this index can be used to fetch the page-directory entry from the top-level page directory. If valid, the second level of the page directory is consulted by combining the physical frame number from the top-level PDE and the

```

1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True) // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10     else // TLB Miss
11         // first, get page directory entry
12         PDIndex = (VPN & PD_MASK) >> PD_SHIFT
13         PDEAddr = PDBR + (PDIndex * sizeof(PDE))
14         PDE = AccessMemory(PDEAddr)
15         if (PDE.Valid == False)
16             RaiseException(SEGMENTATION_FAULT)
17         else
18             // PDE is valid: now fetch PTE from page table
19             PTIndex = (VPN & PT_MASK) >> PT_SHIFT
20             PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
21             PTE = AccessMemory(PTEAddr)
22             if (PTE.Valid == False)
23                 RaiseException(SEGMENTATION_FAULT)
24             else if (CanAccess(PTE.ProtectBits) == False)
25                 RaiseException(PROTECTION_FAULT)
26             else
27                 TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
28             RetryInstruction()

```

Figure 20.6: Multi-level Page Table Control Flow

next part of the VPN (PD Index 1). Finally, if valid, the PTE address can be formed by using the page-table index combined with the address from the second-level PDE. Whew! That's a lot of work. And all just to look something up in a multi-level table.

The Translation Process: Remember the TLB

To summarize the entire process of address translation using a two-level page table, we once again present the control flow in algorithmic form (Figure 20.6). The figure shows what happens in hardware (assuming a hardware-managed TLB) upon *every* memory reference.

As you can see from the figure, before any of the complicated multi-level page table access occurs, the hardware first checks the TLB; upon a hit, the physical address is formed directly *without* accessing the page table at all, as before. Only upon a TLB miss does the hardware need to perform the full multi-level lookup. On this path, you can see the cost of our traditional two-level page table: two additional memory accesses to look up a valid translation.

20.4 Inverted Page Tables

An even more extreme space savings in the world of page tables is found with **inverted page tables**. Here, instead of having many page tables (one per process of the system), we keep a single page table that has an entry for each *physical page* of the system. The entry tells us which process is using this page, and which virtual page of that process maps to this physical page.

Finding the correct entry is now a matter of searching through this data structure. A linear scan would be expensive, and thus a hash table is often built over the base structure to speed up lookups. The PowerPC is one example of such an architecture [JM98].

More generally, inverted page tables illustrate what we've said from the beginning: page tables are just data structures. You can do lots of crazy things with data structures, making them smaller or bigger, making them slower or faster. Multi-level and inverted page tables are just two examples of the many things one could do.

20.5 Swapping the Page Tables to Disk

Finally, we discuss the relaxation of one final assumption. Thus far, we have assumed that page tables reside in kernel-owned physical memory. Even with our many tricks to reduce the size of page tables, it is still possible, however, that they may be too big to fit into memory all at once. Thus, some systems place such page tables in **kernel virtual memory**, thereby allowing the system to **swap** some of these page tables to disk when memory pressure gets a little tight. We'll talk more about this in a future chapter (namely, the case study on VAX/VMS), once we understand how to move pages in and out of memory in more detail.

20.6 Summary

We have now seen how real page tables are built; not necessarily just as linear arrays but as more complex data structures. The trade-offs such tables present are in time and space — the bigger the table, the faster a TLB miss can be serviced, as well as the converse — and thus the right choice of structure depends strongly on the constraints of the given environment.

In a memory-constrained system (like many older systems), small structures make sense; in a system with a reasonable amount of memory and with workloads that actively use a large number of pages, a bigger table that speeds up TLB misses might be the right choice. With software-managed TLBs, the entire space of data structures opens up to the delight of the operating system innovator (hint: that's you). What new structures can you come up with? What problems do they solve? Think of these questions as you fall asleep, and dream the big dreams that only operating-system developers can dream.

References

- [BOH10] “Computer Systems: A Programmer’s Perspective” by Randal E. Bryant and David R. O’Hallaron. Addison-Wesley, 2010. *We have yet to find a good first reference to the multi-level page table. However, this great textbook by Bryant and O’Hallaron dives into the details of x86, which at least is an early system that used such structures. It’s also just a great book to have.*
- [JM98] “Virtual Memory: Issues of Implementation” by Bruce Jacob, Trevor Mudge. IEEE Computer, June 1998. *An excellent survey of a number of different systems and their approach to virtualizing memory. Plenty of details on x86, PowerPC, MIPS, and other architectures.*
- [LL82] “Virtual Memory Management in the VAX/VMS Operating System” by Hank Levy, P. Lipman. IEEE Computer, Vol. 15, No. 3, March 1982. *A terrific paper about a real virtual memory manager in a classic operating system, VMS. So terrific, in fact, that we’ll use it to review everything we’ve learned about virtual memory thus far a few chapters from now.*
- [M28] “Reese’s Peanut Butter Cups” by Mars Candy Corporation. Published at stores near you. *Apparently these fine confections were invented in 1928 by Harry Burnett Reese, a former dairy farmer and shipping foreman for one Milton S. Hershey. At least, that is what it says on Wikipedia. If true, Hershey and Reese probably hate each other’s guts, as any two chocolate barons should.*
- [N+02] “Practical, Transparent Operating System Support for Superpages” by Juan Navarro, Sitaram Iyer, Peter Druschel, Alan Cox. OSDI ’02, Boston, Massachusetts, October 2002. *A nice paper showing all the details you have to get right to incorporate large pages, or *superpages*, into a modern OS. Not as easy as you might think, alas.*
- [M07] “Multics: History” Available: <http://www.multicians.org/history.html>. *This amazing web site provides a huge amount of history on the Multics system, certainly one of the most influential systems in OS history. The quote from therein: “Jack Dennis of MIT contributed influential architectural ideas to the beginning of Multics, especially the idea of combining paging and segmentation.” (from Section 1.2.1)*

Homework (Simulation)

This fun little homework tests if you understand how a multi-level page table works. And yes, there is some debate over the use of the term “fun” in the previous sentence. The program is called, perhaps unsurprisingly: `paging-multilevel-translate.py`; see the README for details.

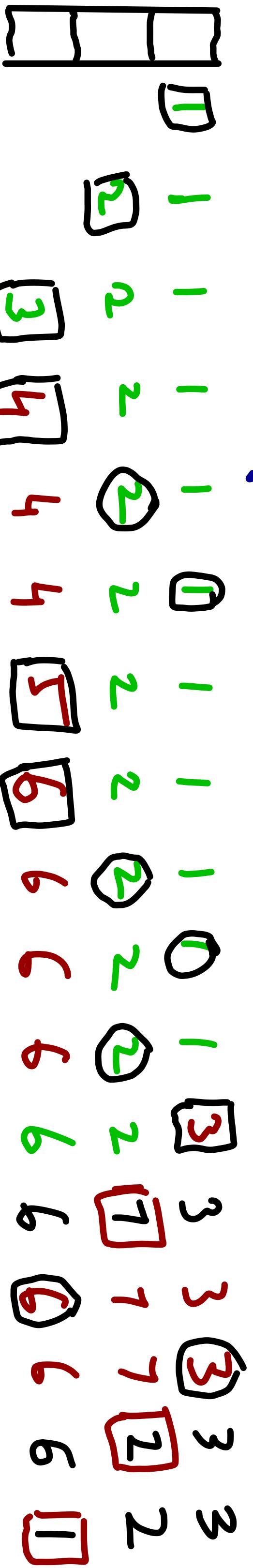
Questions

1. With a linear page table, you need a single register to locate the page table, assuming that hardware does the lookup upon a TLB miss. How many registers do you need to locate a two-level page table? A three-level table?
2. Use the simulator to perform translations given random seeds 0, 1, and 2, and check your answers using the `-c` flag. How many memory references are needed to perform each lookup?
3. Given your understanding of how cache memory works, how do you think memory references to the page table will behave in the cache? Will they lead to lots of cache hits (and thus fast accesses?) Or lots of misses (and thus slow accesses)?

Consider following page reference

1 2 3 4 2 1 5 6 2 1 2 3 7 6 3 2 1 2 3 6

How many page + mult occurr? consider 3 frames



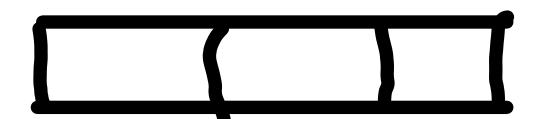
Page fault (*) = "hit ratio 0"

Page hit (\uparrow) = 95%

•
↑ ↑ *
1 - 2 3
2 2 3
2 2 6
* ↑ ↑ *
↑ ↑ *
* ↑ ↑ *
* ↑ ↑ *
* ↑ ↑ *
* ↑ ↑ *
* ↑ ↑ *
* ↑ ↑ *
* ↑ ↑ *
* ↑ ↑ *
* ↑ ↑ *
* ↑ ↑ *
* ↑ ↑ *
* ↑ ↑ *

Optimal

#IFO



- 2 3 4 2 1 5 6 2 1 2 3 6

- - - - - - - - - - - - - - - -

2

2

2

2

2

2

2

2

2

2

2

2

3

3

3

3

3

3

3

3

3

3

3

* * * * * * * * * * * * * * * *

Page fault (*) = 16

Hit ratio = 16/20 = 0.8

20%

Page hit (%) = 4

80%

LRU

1 2 3 4 2 1 5 6 2 - 2 3 7 6 3 2 - 2 3 6



=

2 -

2 -

4

2 4

5

2 5

5

6 5

7

3 7

2

3 2

0

*

*

*

*

→

*

*

*

*

↑

*

*

*

→

1 - 2
x - 3
- 16

miss ratio = $\frac{15}{26} = 0.5769\ldots$

+

Memory Allocation Algorithms

Variable sized allocation

- Given a block of memory, how do we allocate it to satisfy various memory allocation requests?
- This problem must be solved in the C library
 - Allocates one or more pages from kernel via `brk`/`sbrk` or `mmap` system calls
 - Gives out smaller chunks to user programs via `malloc`
- This problem also occurs in the kernel
 - Kernel must allocate memory for its internal data structures

Variable sized allocation: headers

- Consider a simple implementation of malloc
- Every allocated chunk has a header with info like size of chunk
 - Why store size? We should know how much to free when `free()` is called

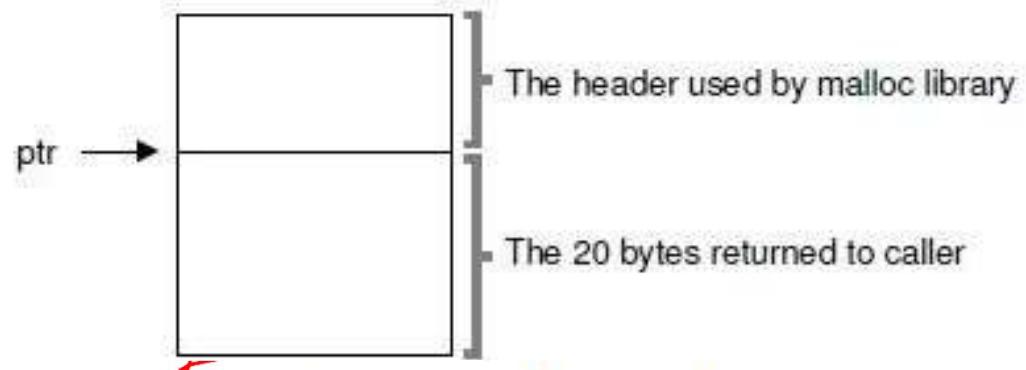


Figure 17.1: An Allocated Region Plus Header

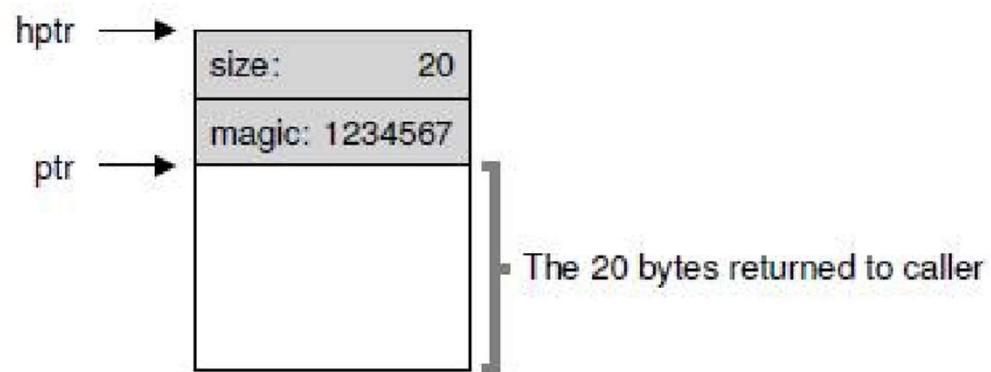


Figure 17.2: Specific Contents Of The Header

Free list

- Free space is managed as a list
 - Pointer to the next free chunk is embedded within the free chunk
- The library tracks the head of the list
 - Allocations happen from the head

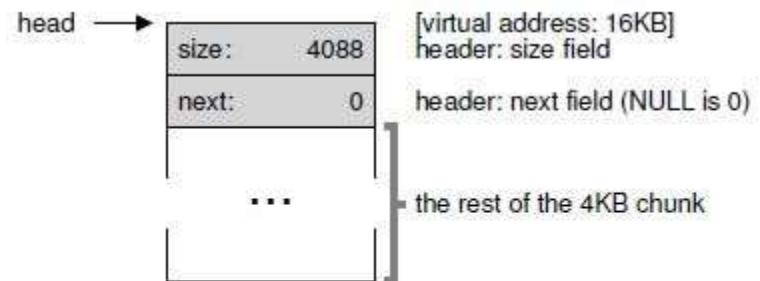


Figure 17.3: A Heap With One Free Chunk

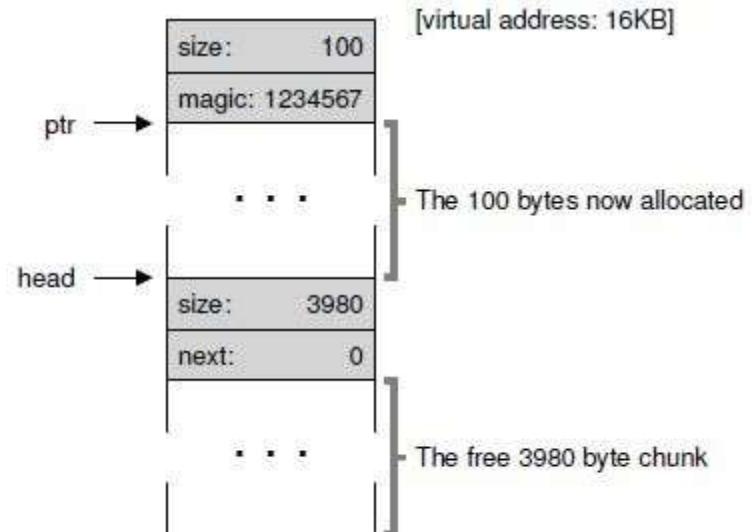
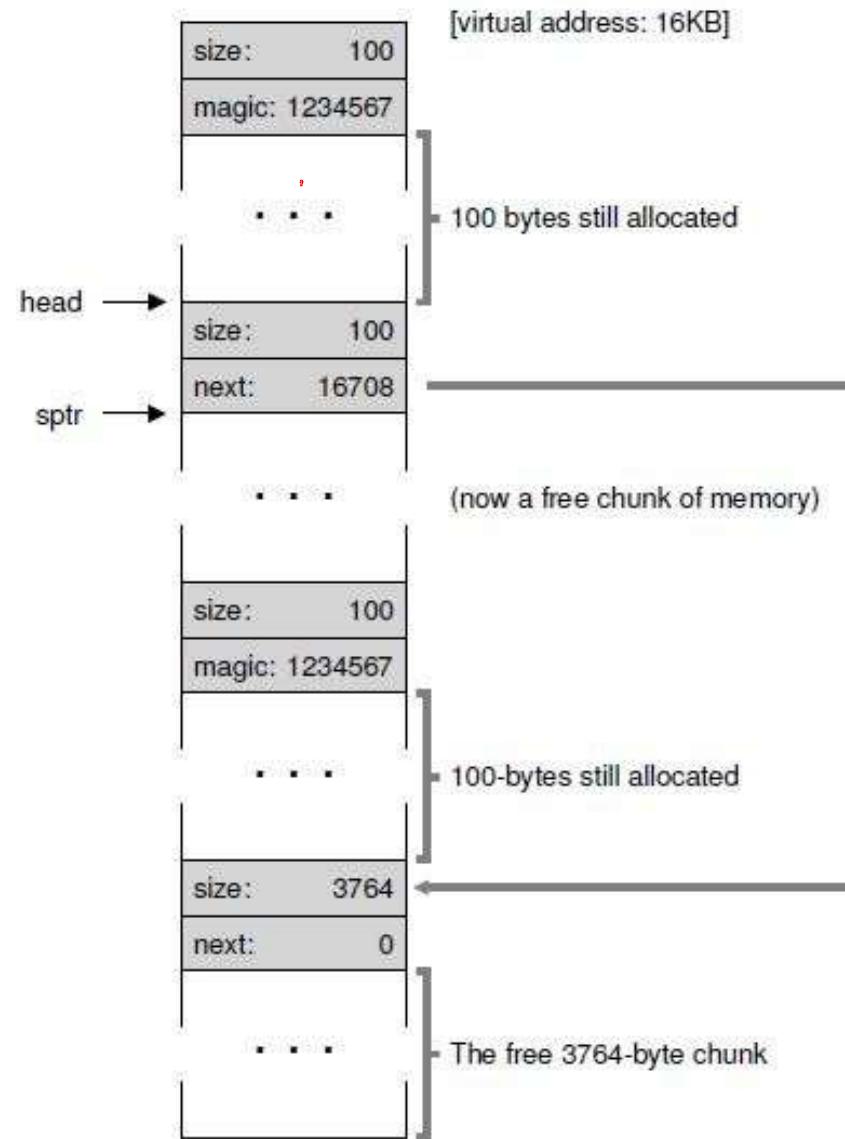


Figure 17.4: A Heap: After One Allocation

External fragmentation

- Suppose 3 allocations of size 100 bytes each happen. Then, the middle chunk pointed to by `sptr` is freed
- What is the free list?
 - It now has two non-contiguous elements
- Free space may be scattered around due to fragmentation
 - Cannot satisfy a request for 3800 bytes even though we have the free space



Splitting and Coalescing

- Suppose all the three chunks are freed
- The list now has a bunch of free chunks that are adjacent
- A smart algorithm would merge them all into a bigger free chunk
- Must split and coalesce free chunks to satisfy variable sized requests

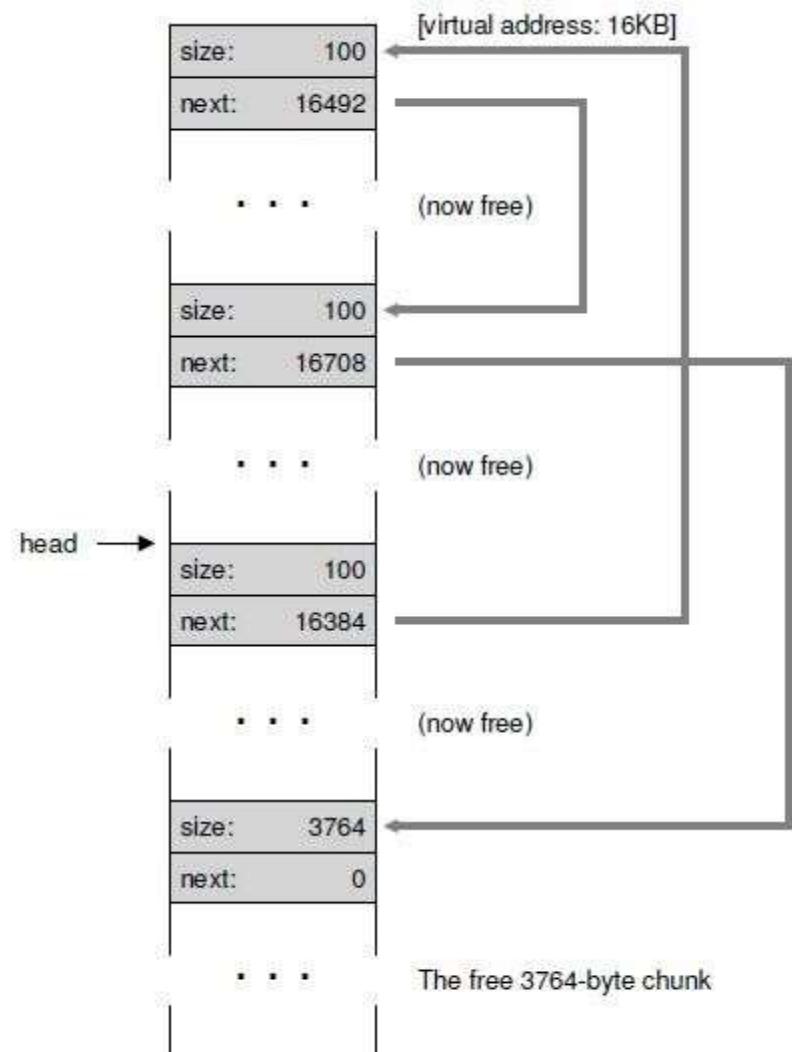
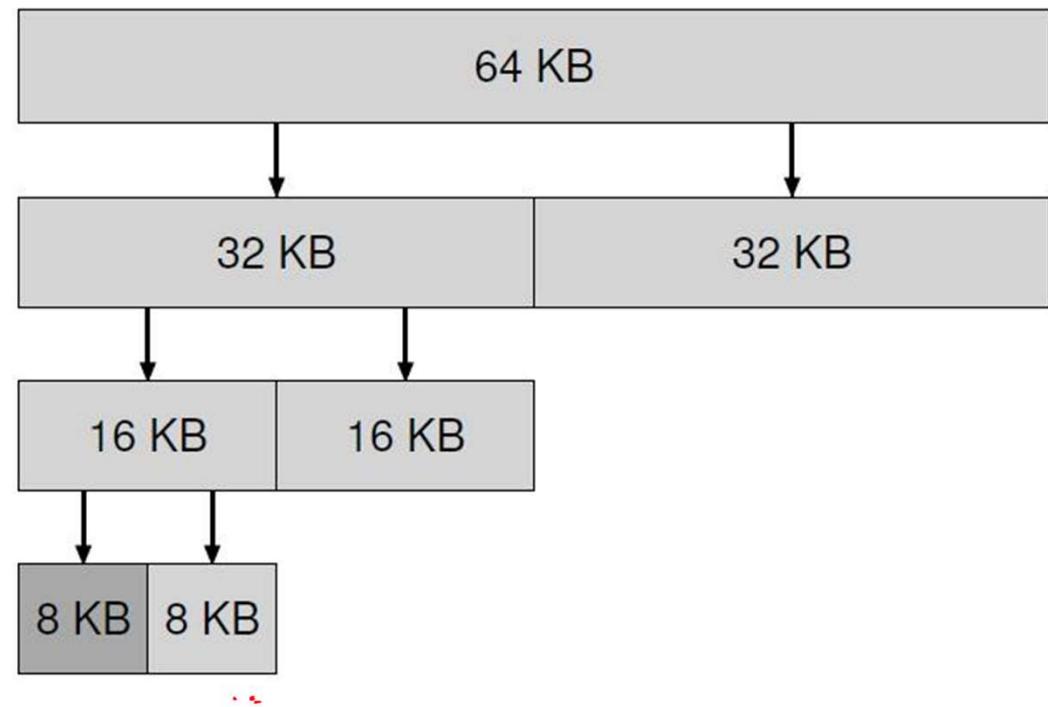


Figure 17.7: A Non-Coalesced Free List

Buddy allocation for easy coalescing

- Allocate memory in size of power of 2
 - E.g., for a request of 7000 bytes, allocate 8 KB chunk
- Why? 2 adjacent power-of-2 chunks can be merged to form a bigger power-of-2 chunk
 - E.g., if 8KB block and its “buddy” are free, they can form a 16KB chunk



Variable Size Allocation Strategies

- First fit: allocate first free chunk that is sufficient
- Best fit: allocate free chunk that is closest in size
- Worst fit: allocate free chunk that is farthest in size
- Example, consider this free list, and malloc(15)



- Best fit would allocate the 20-byte chunk



- Worst fit would allocate 30-byte chunk: remaining chunk is bigger and more usable



Fixed size allocations

- Memory allocation algorithms are much simpler with fixed size allocations
- Page-sized fixed allocations in kernel:
 - Has free list of pages
 - Pointer to next page stored in the free page itself
- For some smaller allocations (e.g., PCB), kernel uses a slab allocator
 - Object caches for each type (size) of objects
 - Within each cache, only fixed size allocation
 - Each cache is made up of one or more “slabs”
- Fixed size memory allocators can be used in user programs also (instead of generic malloc)

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 details¹.

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

¹It is nearly 80 pages long; thus, you really have to be interested!

would be useful to combat fragmentation². 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

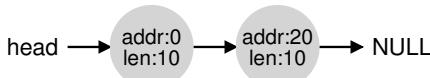
Before delving into some policy details, we'll first cover some common mechanisms used in most allocators. First, we'll discuss the basics of splitting and coalescing, common techniques in most any allocator. Second, we'll show how one can track the size of allocated regions quickly and with relative 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

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):

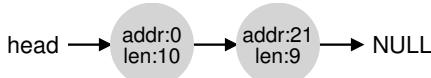


As described above, a request for anything greater than 10 bytes will fail (returning `NULL`); there just isn't a single contiguous chunk of memory of that size available. A request for exactly that size (10 bytes) could be satisfied easily by either of the free chunks. But what happens if the request 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

²Once 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 9³. 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.

³This discussion assumes that there are no headers, an unrealistic but simplifying assumption we make for now.

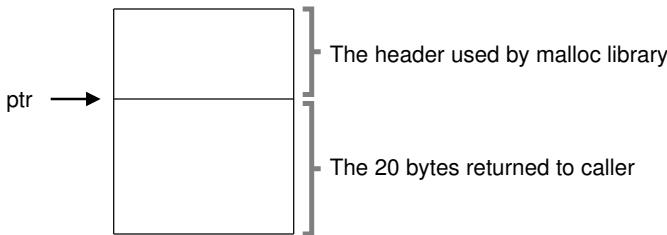


Figure 17.1: An Allocated Region Plus Header

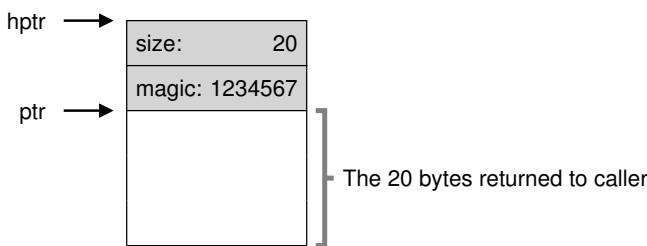


Figure 17.2: Specific Contents Of The Header

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 {
    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 = (header_t *) ptr - 1;
    ...
}
```

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;
```

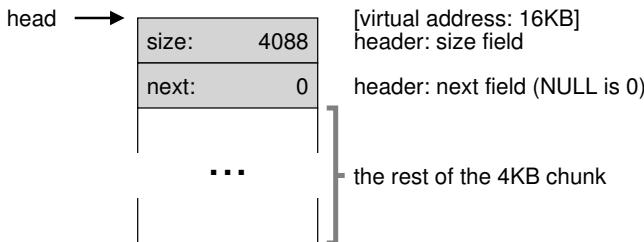


Figure 17.3: A Heap With One Free Chunk

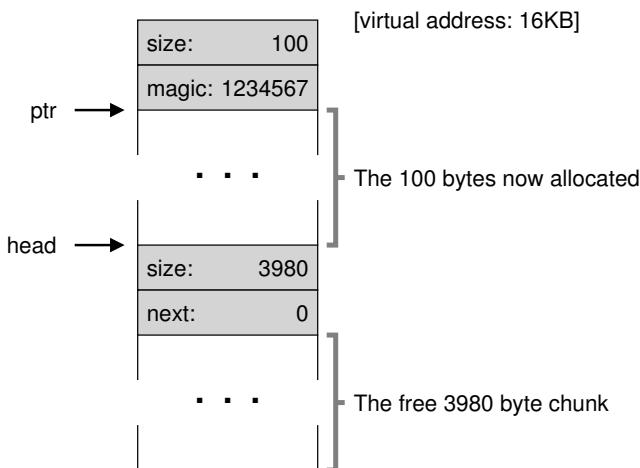


Figure 17.4: A Heap: After One Allocation

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 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

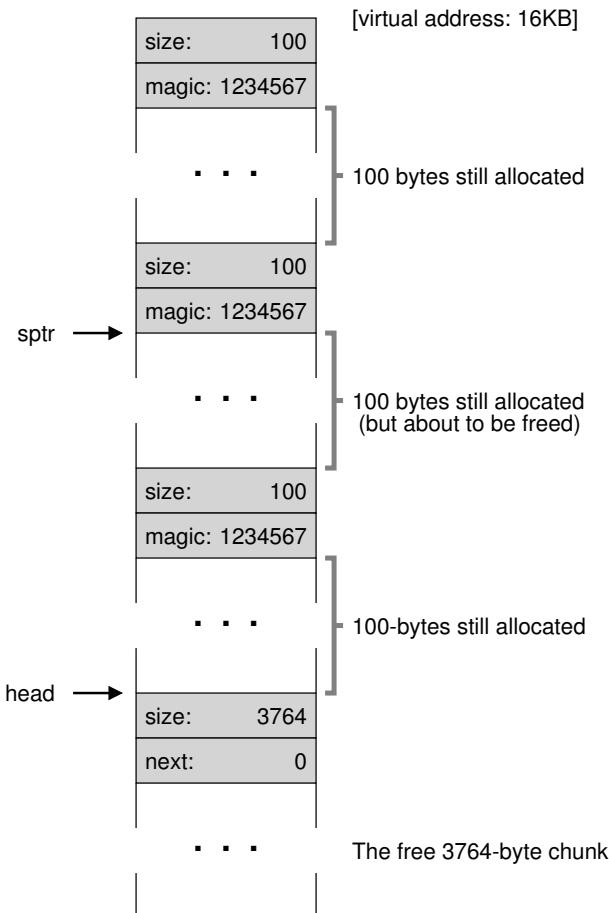


Figure 17.5: Free Space With Three Chunks Allocated

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 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

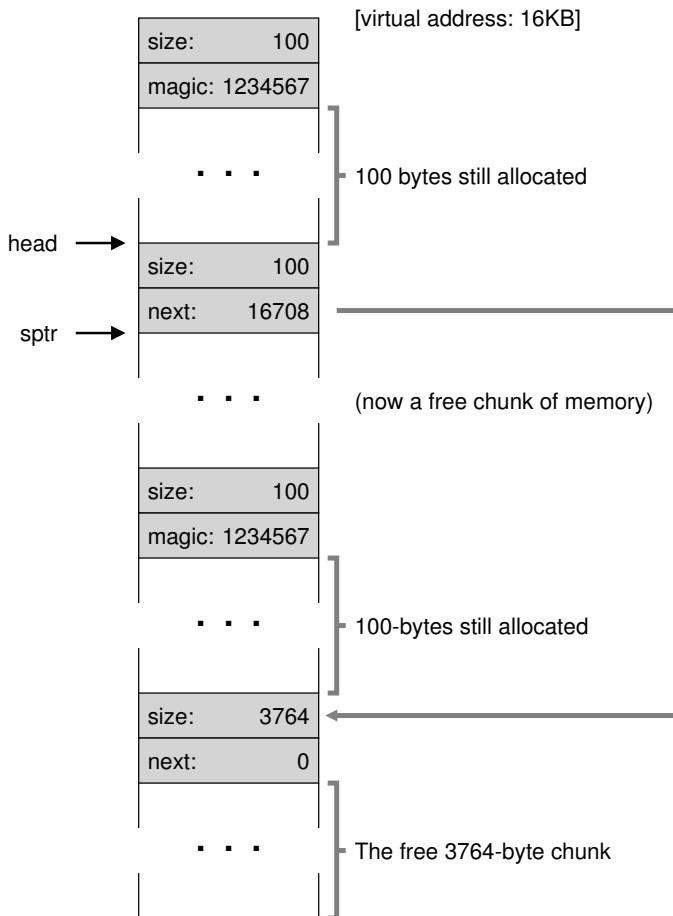


Figure 17.6: Free Space With Two Chunks Allocated

bytes in size after the three splits. But what happens when the calling program returns some memory via `free()`?

In this example, the application returns the middle chunk of allocated memory, by calling `free(16500)` (the value 16500 is arrived upon by adding the start of the memory region, 16384, to the 108 of the previous chunk and the 8 bytes of the header for this chunk). This value is shown in the previous diagram by the pointer `sptr`.

The library immediately figures out the size of the free region, and then adds the free chunk back onto the free list. Assuming we insert at the head of the free list, the space now looks like this (Figure 17.6).

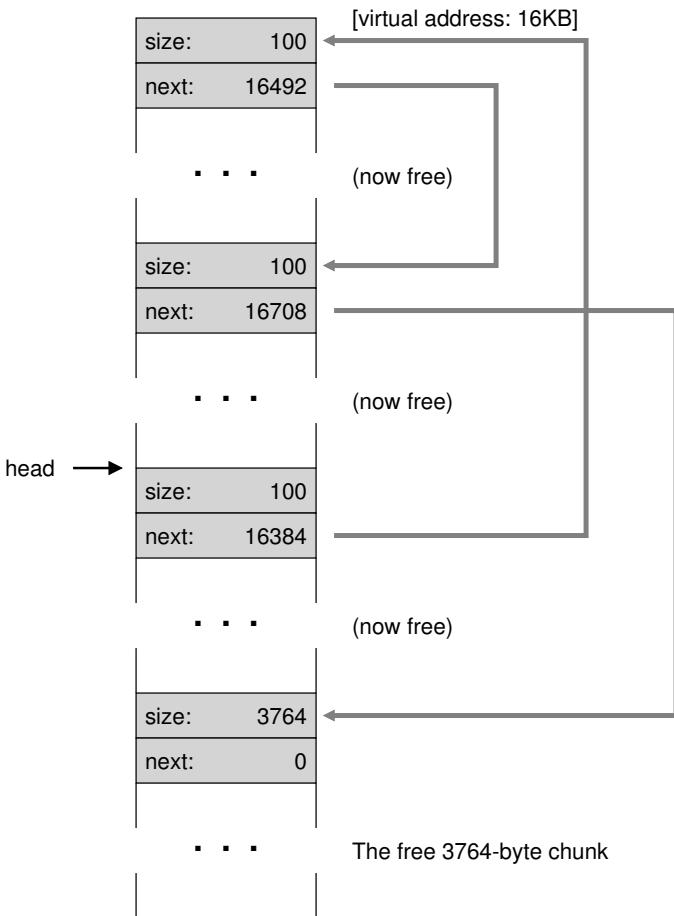


Figure 17.7: A Non-Coalesced Free List

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). 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 end up with fragmentation (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 (Figure 17.8, page 15).

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.

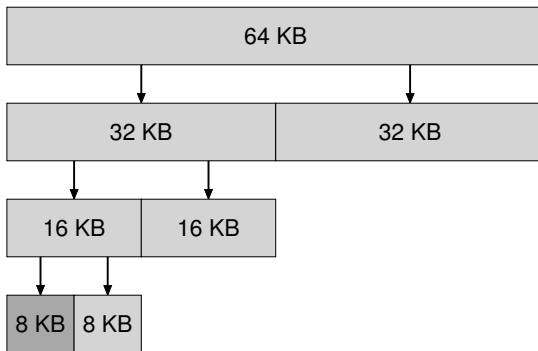


Figure 17.8: Example Buddy-managed Heap

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" by Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, 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" by 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" by Jason Evans. April, 2006. <http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf>. *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" by Kenneth C. Knowlton. Communications of the ACM, Volume 8:10, 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 software⁴.*
- [S15] "Understanding glibc malloc" by Sploitfun. February, 2015. 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" by Paul R. Wilson, Mark S. Johnstone, Michael Neely, David Boles. International Workshop on Memory Management, Scotland, UK, 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!*

⁴Actually we use LaTeX, which is based on Lamport's additions to TeX, but close enough.

Homework (Simulation)

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.

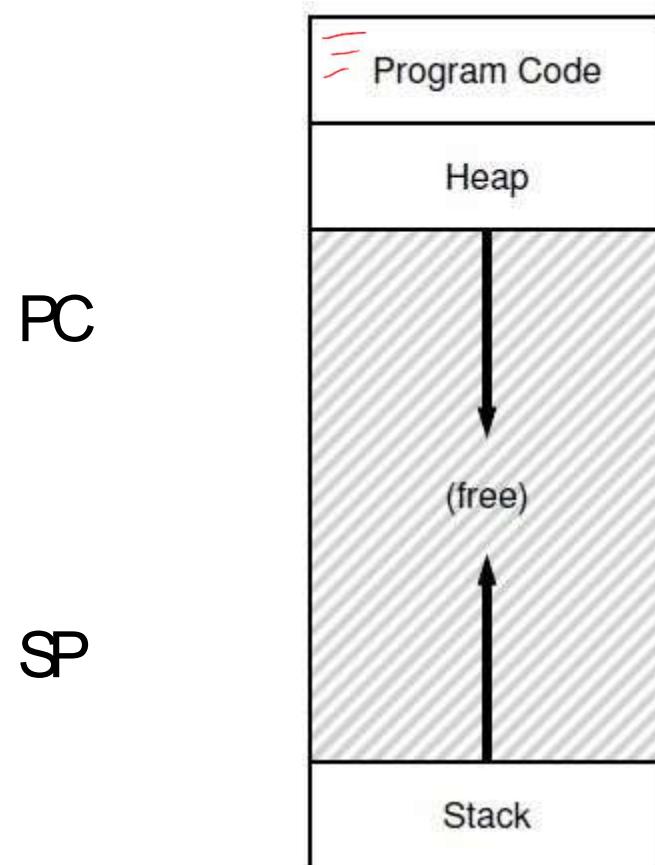
Questions

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 (`-1 ADDRSORT, -1 SIZESORT+, -1 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 the percent 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.

Threads and Concurrency

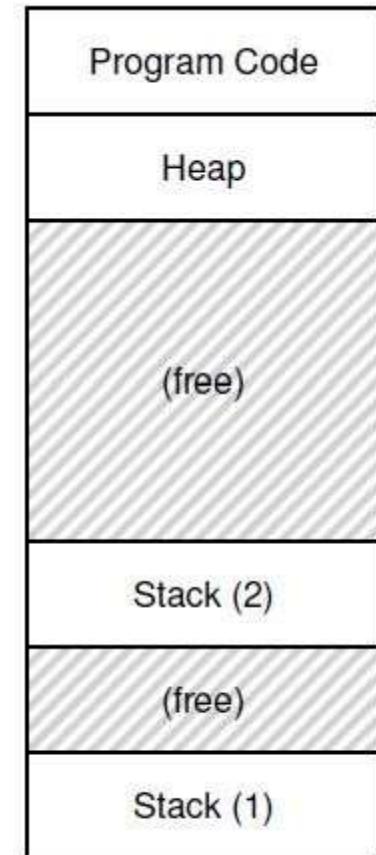
Single threaded process

- So, far we have studied single threaded programs
- Recap: process execution
 - PCpoints to current instruction being run
 - SPpoints to stack frame of current function call
- A program can also have multiple threads of execution
- What is a thread?



Multi threaded process

- A thread is like another copy of a process that executes independently
 - PC1
 - PC2
- Threads share the same address space (code, heap)
- Each thread has separate PC
 - Each thread may run over different part of the program
- Each thread has separate stack for independent function calls
 - SP2
 - SP1



Process vs. threads

- Parent Pforks a child C
 - P and C do not share any memory
 - Need complicated IPC mechanisms to communicate
 - Extra copies of code, data in memory
- Parent Pexecutes two threads T1 and T2
 - T1 and T2 share parts of the address space
 - Global variables can be used for communication
 - Smaller memory footprint
- Threads are like separate processes, except they share the same address space

Why threads?

- Parallelism: a single process can effectively utilize multiple CPU cores
 - Understand the difference between concurrency and parallelism
 - Concurrency: running multiple threads/processes at the same time, even on single CPU core, by interleaving their executions
 - Parallelism: running multiple threads/processes in parallel over different CPU cores
- Even if no parallelism, concurrency of threads ensures effective use of CPU when one of the threads blocks (e.g., for I/O)

Scheduling threads

- OS schedules threads that are ready to run independently, much like processes
- The context of a thread (PC, registers) is saved into/restored from thread control block(TCB)
 - Every PCB has one or more linked TCBs
- Threads that are scheduled independently by kernel are called kernel threads
 - E.g., Linux pthreads are kernel threads

Creating threads using pthreads API

```
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 ...
12 main(int argc, char *argv[]) {
13     pthread_t p1, p2;
14     int rc;
15     printf("main: begin\n");
16     rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
17     rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
18     // join waits for the threads to finish
19     rc = pthread_join(p1, NULL); assert(rc == 0);
20     rc = pthread_join(p2, NULL); assert(rc == 0);
21     printf("main: end\n");
22 }
```

Figure 26.2: Simple Thread Creation Code (t0.c)

Example: threads with shared data

```
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    }
```

Threads with shared data: what happens?

- What do we expect? Two threads, each increments counter by 10^7 , so 2×10^7

```
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)
```

- Sometimes, a lower value. Why?

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

What is happening?

- Assembly code of
counter = counter + 1

```
100 mov    0x8049a1c, %eax  
105 add    $0x1, %eax  
108 mov    %eax, 0x8049a1c
```

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
interrupt					
	<i>save T1's state</i>				
	<i>restore T2's state</i>		100	0	50
			105	50	50
			108	51	50
			113	51	51
interrupt					
	<i>save T2's state</i>				
	<i>restore T1's state</i>		108	51	51
			113	51	51

Figure 26.7: The Problem: Up Close and Personal

Race conditions and synchronization

- What just happened is called a race condition
 - Concurrent execution can lead to different results
- Critical section: portion of code that can lead to race conditions
- What we need: mutual exclusion
 - Only one thread should be executing critical section at any time
- What we need: atomicity of the critical section
 - The critical section should execute like one uninterruptible instruction
- How is it achieved? Locks

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

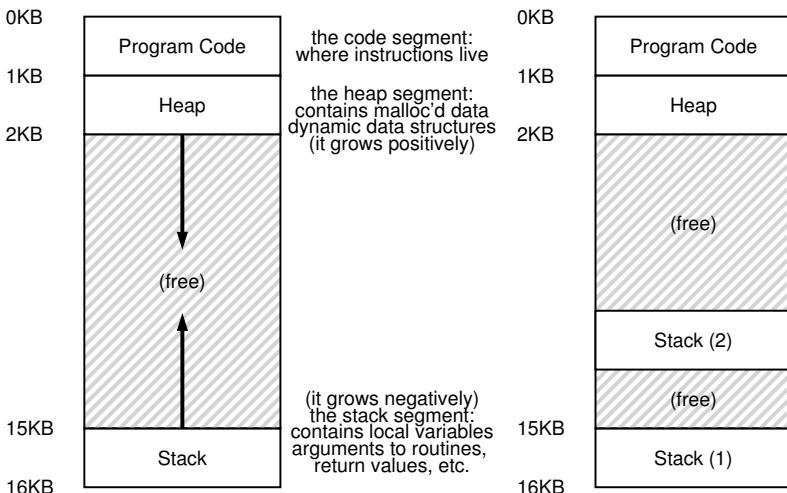


Figure 26.1: Single-Threaded And Multi-Threaded Address Spaces

(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).

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.

```

1 #include <stdio.h>
2 #include <assert.h>
3 #include <pthread.h>
4 #include "common.h"
5 #include "common_threads.h"
6
7 void *mythread(void *arg) {
8     printf("%s\n", (char *) arg);
9     return NULL;
10 }
11
12 int
13 main(int argc, char *argv[]) {
14     pthread_t p1, p2;
15     int rc;
16     printf("main: begin\n");
17     Pthread_create(&p1, NULL, mythread, "A");
18     Pthread_create(&p2, NULL, mythread, "B");
19     // join waits for the threads to finish
20     Pthread_join(p1, NULL);
21     Pthread_join(p2, NULL);
22     printf("main: end\n");
23     return 0;
24 }
```

Figure 26.2: Simple Thread Creation Code (`t0.c`)

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.

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 6) 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

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)

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		
	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)

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 7).

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 (10^7) 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; ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include "common.h"
4 #include "common_threads.h"
5
6 static volatile int counter = 0;
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 *mythread(void *arg) {
15     printf("%s: begin\n", (char *) arg);
16     int i;
17     for (i = 0; i < 1e7; i++) {
18         counter = counter + 1;
19     }
20     printf("%s: done\n", (char *) arg);
21     return NULL;
22 }
23
24 // main()
25 //
26 // Just launches two threads (pthread_create)
27 // and then waits for them (pthread_join)
28 //
29 int main(int argc, char *argv[]) {
30     pthread_t p1, p2;
31     printf("main: begin (counter = %d)\n", counter);
32     Pthread_create(&p1, NULL, mythread, "A");
33     Pthread_create(&p2, NULL, mythread, "B");
34
35     // join waits for the threads to finish
36     Pthread_join(p1, NULL);
37     Pthread_join(p2, NULL);
38     printf("main: done with both (counter = %d)\n",
39             counter);
40     return 0;
41 }
```

Figure 26.6: Sharing Data: Uh Oh (**t1.c**)

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 `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):

OS	Thread 1	Thread 2	(after instruction)		
			PC	eax	counter
	<i>before critical section</i>		100	0	50
	mov 8049a1c, %eax		105	50	50
	add \$0x1, %eax		108	51	50
interrupt					
	<i>save T1</i>				
	<i>restore T2</i>				
		mov 8049a1c, %eax	100	0	50
		add \$0x1, %eax	105	50	50
		mov %eax, 8049a1c	108	51	50
interrupt					
	<i>save T2</i>				
	<i>restore T1</i>				
		mov %eax, 8049a1c	108	51	51
			113	51	51

Figure 26.7: The Problem: Up Close and Personal

```

100 mov    0x8049a1c, %eax
105 add    $0x1, %eax
108 mov    %eax, 0x8049a1c

```

With these assumptions, what happens is shown in Figure 26.7 (page 10). 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** (or, more specifically, a **data race**): 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 **indefinite**, 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.

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.

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 this hardware support, in combination with some help from the operating system, we will be able to build multi-threaded code that accesses critical sections in a 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 SUPPORT 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.

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** (or **data race** [NM92]) arises if multiple threads of execution enter the critical section at roughly the same time; both attempt to update the shared data structure, leading to a surprising (and perhaps undesirable) outcome.
- An **indeterminate** program consists of one or more race conditions; the output of the program varies from run to run, depending on which threads ran when. The outcome is thus not **deterministic**, something we usually expect from computer systems.
- To avoid these problems, threads should use some kind of **mutual exclusion** primitives; doing so guarantees that only a single thread ever enters a critical section, thus avoiding races, and resulting in deterministic program outputs.

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.

References

- [D65] “Solution of a problem in concurrent programming control” by 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” by Edsger W. Dijkstra. 1968. Available at this site: <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” by 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” by 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.*
- [NM92] “What Are Race Conditions? Some Issues and Formalizations” by Robert H. B. Netzer and Barton P. Miller. ACM Letters on Programming Languages and Systems, Volume 1:1, March 1992. *An excellent discussion of the different types of races found in concurrent programs. In this chapter (and the next few), we focus on data races, but later we will broaden to discuss general races as well.*
- [SR05] “Advanced Programming in the UNIX Environment” by 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 (Simulation)

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, then answer the questions below.

Questions

1. Let's examine a simple program, "loop.s". First, just read and understand it. 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`. What will `%dx` be during the run? Use the `-c` flag to check your answers; the answers, on the left, show the value of the register (or memory value) *after* the instruction on the right has run.
 `.386`
 `.model small`
 `.stack 4096`
 `.data`
 `dx dw 0`
 `.code`
 `mov ax, dx` *(initial value)*
 `add dx, 1` *(instruction)*
 `jne .` *(final value)*
2. Same code, different 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` to 3. What values will `%dx` see? Run with `-c` to check. Does the presence of multiple threads affect your calculations? Is there a race in this code?
3. Run this: `./x86.py -p loop.s -t 2 -i 3 -r -a dx=3, dx=3 -R dx` This makes the interrupt interval small/random; use different seeds (`-s`) to see different interleavings. Does the interrupt frequency change anything?
4. Now, a different program, `looping-race-nolock.s`, which accesses a shared variable located at address 2000; we'll call this variable `value`. Run it with a single thread to confirm your understanding: `./x86.py -p looping-race-nolock.s -t 1 -M 2000` What is `value` (i.e., at memory address 2000) throughout the run? Use `-c` to check.
5. Run with multiple iterations/threads: `./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000` Why does each thread loop three times? What is final value of `value`?
6. Run with random interrupt intervals: `./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0` with different seeds (`-s 1, -s 2, etc.`) Can you tell by looking at the thread interleaving what the final value of `value` will be? Does the timing of the interrupt matter? Where can it safely occur? Where not? In other words, where is the critical section exactly?

7. Now examine fixed interrupt intervals: `./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1` What will the final value of the shared variable `value` be? What about when you change `-i 2`, `-i 3`, etc.? For which interrupt intervals does the program give the “correct” answer?
8. Run the same for more loops (e.g., set `-a bx=100`). What interrupt intervals (`-i`) lead to a correct outcome? Which intervals are surprising?
9. One last program: `wait-for-me.s`. Run: `./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 `%ax` and memory location 2000. 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?

Locks

Locks: Basic idea

- Consider update of shared variable

```
balance = balance + 1;
```

- We can use a special lock variable to protect it

```
1  lock_t mutex; // some globally-allocated lock 'mutex'  
2  ...  
3  lock(&mutex);  
4  balance = balance + 1;  
5  unlock(&mutex);
```

- All threads accessing a critical section share a lock
- One threads succeeds in locking – owner of lock
- Other threads that try to lock cannot proceed further until lock is released by the owner
- Pthreads library in Linux provides such locks

Building a lock

- Goals of a lock implementation
 - Mutual exclusion (obviously!)
 - Fairness: all threads should eventually get the lock, and no thread should starve
 - Low overhead: acquiring, releasing, and waiting for lock should not consume too many resources
- Implementation of locks are needed for both userspace programs (e.g., pthreads library) and kernel code
- Implementing locks needs support from hardware and OS

Is disabling interrupts enough?

- Is this enough?
- No, not always!
- Many issues here:
 - Disabling interrupts is a privileged instruction and program can misuse it (e.g., run forever)
 - Will not work on multiprocessor systems, since another thread on another core can enter critical section
- This technique is used to implement locks on single processor systems inside the OS
 - Need better solution for other situations

```
1 void lock() {  
2     DisableInterrupts();  
3 }  
4 void unlock() {  
5     EnableInterrupts();  
6 }
```

A failed lock implementation (1)

- Lock: spin on a flag variable until it isunset, then set it to acquire lock
- Unlock: unset flag variable

```
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

A failed lock implementation (2)

- Thread 1 spins, lock is released, ends spin
- Thread 1 interrupted just before setting flag
- Race condition has moved to the lock acquisition code!

Thread 1	Thread 2
call lock () while (flag == 1) interrupt: switch to Thread 2	
	call lock () while (flag == 1) flag = 1; interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Figure 28.2: Trace: No Mutual Exclusion

Solution: Hardware atomic instructions

- Very hard to ensure atomicity only in software
- Modern architectures provide hardware atomic instructions
- Example of an atomic instruction: test-and-set
 - Update a variable and return old value, all in one hardware instruction

```
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     }
```

Simple lock using test-and-set

- If TestAndSet(flag,1) returns 1, it means the lock is held by someone else, so wait busily
- This lock is called a spinlock – spins until lock is acquired

```
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

Spinlock using compare-and-swap

- Another atomic instruction: compare-and-swap

```
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

```
1 void lock(lock_t *lock) {  
2     while (CompareAndSwap(&lock->flag, 0, 1) == 1)  
3         ; // spin  
4 }
```

Alternative to spinning

- Alternative to spinlock: a (sleeping) mutex
- Instead of spinning for a lock, a contending thread could simply give up the CPU and check back later
 - `yield()` moves thread from running to ready state

```
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

Spinlock vs. sleeping mutex

- Most userspace lock implementations are of the sleeping mutex kind
 - CPU wasted by spinning contending threads
 - More so if a thread holds spinlock and blocks for long
- Locks inside the OS are always spinlocks
 - Why? Who will the OS yield to?
- When OS acquires a spinlock:
 - It must disable interrupts (on that processor core) while the lock is held. Why? An interrupt handler could request the same lock, and spin for it forever.
 - It must not perform any blocking operation – never go to sleep with a locked spinlock!
- In general, use spinlocks with care, and release as soon as possible

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; exits on failure
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 provide 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 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.

```
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

For these reasons, turning off interrupts is only used in limited contexts as a mutual-exclusion primitive. For example, in some cases an 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.

28.6 A Failed Attempt: Just Using Loads/Stores

To move beyond interrupt-based techniques, we will have to rely on CPU hardware and the instructions it provides us to build a proper lock. Let's first try to build a simple lock by using a single flag variable. In this failed attempt, we'll see some of the basic ideas needed to build a lock, and (hopefully) see why just using a single variable and accessing it via normal loads and stores is insufficient.

In this first attempt (Figure 28.1), the idea is quite simple: use a simple variable (`flag`) 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 an-

Thread 1	Thread 2
call lock()	
while (flag == 1)	
interrupt: switch to Thread 2	
	call lock()
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Figure 28.2: Trace: No Mutual Exclusion

other 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; 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.

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 Working Spin Locks with Test-And-Set

Because disabling interrupts does not work on multiple processors, and because simple approaches using loads and stores (as shown above) don’t work, 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 known as a **test-and-set** (or **atomic exchange**¹) instruction. We define what the test-and-set instruction does via 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 }
```

¹Each architecture that supports test-and-set calls it by a different name. On SPARC it is called the load/store unsigned byte instruction (`ldstb`); on x86 it is the locked version of the atomic exchange (`xchg`).

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() {
    // indicate you intend to hold the lock w/ 'flag'
    flag[0] = flag[1] = 0;
    // whose turn is it? (thread 0 or 1)
    turn = 0;
}
void lock() {
    // 'self' is the thread ID of caller
    flag[self] = 1;
    // make it other thread's turn
    turn = 1 - self;
    while ((flag[1-self] == 1) && (turn == 1 - self))
        ; // spin-wait while it's not your turn
}
void unlock() {
    // simply undo your intent
    flag[self] = 0;
}
```

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...

```

1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0: lock is available, 1: lock 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

What the test-and-set instruction does is as follows. It returns the old value pointed to by the `old_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)`, 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

TIP: THINK ABOUT CONCURRENCY AS A 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)

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 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. Simple spin locks (as discussed thus far) 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 threads spread out across many CPUs.

For spin locks, in the single CPU case, performance overheads can be quite painful; imagine the case where the thread holding the lock is preempted 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

```

1 int CompareAndSwap(int *ptr, int expected, int new) {
2     int original = *ptr;
3     if (original == expected)
4         *ptr = new;
5     return original;
6 }
```

Figure 28.4: Compare-and-swap

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 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 original 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, the code sequence (from [S05]) might be useful².

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

²github.com/remzi-arpacidusseau/ostep-code/tree/master/threads-locks

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.

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.

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,

```

1 int LoadLinked(int *ptr) {
2     return *ptr;
3 }
4
5 int StoreConditional(int *ptr, int value) {
6     if (no update to *ptr since 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**

```

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

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) ||
3             !StoreConditional(&lock->flag, 1))
4         ; // spin
5 }
```

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 }
```

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.

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 is found in Figure 28.7 (page 14).

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,

```

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

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 (page 15) presents the approach.

```
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

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 **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.

```

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
33                                     // (for next thread!)
34     m->guard = 0;
35 }
```

Figure 28.9: Lock With Queues, Test-and-set, Yield, And Wakeup

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.

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.

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

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.

You might also observe 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()` function 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 further detect 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.

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

```

1 void mutex_lock (int *mutex) {
2     int v;
3     /* Bit 31 was clear, we got the mutex (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 waitFirst make sure the futex value
13           we are monitoring is truly negative (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 counter results in 0 if and
23        only if 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

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 something³.

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].

³Like 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)

References

- [D91] "Just Win, Baby: Al Davis and His Raiders" by Glenn Dickey. Harcourt, 1991. *The book about Al Davis and his famous quote. Or, we suppose, the book is more about Al Davis and the Raiders, and not so much the quote. To be clear: we are not recommending this book, we just needed a citation.*
- [D+13] "Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask" by Tudor David, Rachid Guerraoui, Vasileios Trigonakis. SOSP '13, Nemacolin Woodlands Resort, Pennsylvania, November 2013. *An excellent paper comparing many different ways to build locks using hardware primitives. Great to see how many ideas work on modern hardware.*
- [D68] "Cooperating sequential processes" by Edsger W. Dijkstra. 1968. Available online here: <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF>. *One of the early seminal papers. Discusses how Dijkstra posed the original concurrency problem, and Dekker's solution.*
- [H93] "MIPS R4000 Microprocessor User's Manual" by Joe Heinrich. Prentice-Hall, June 1993. Available: <http://cag.csail.mit.edu/raw/documents/R4400.Uman.book.Ed2.pdf>. *The old MIPS user's manual. Download it while it still exists.*
- [H91] "Wait-free Synchronization" by Maurice Herlihy. ACM TOPLAS, Volume 13: 1, January 1991. *A landmark paper introducing a different approach to building concurrent data structures. Because of the complexity involved, some of these ideas have been slow to gain acceptance in deployment.*
- [L81] "Observations on the Development of an Operating System" by 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)" by Many authors.. Available here: <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?" by A. Mayer. 1982. Available: www.ajwm.net/amayer/papers/B5000.html. *"It (RDLK) is an indivisible operation which reads from and writes into a memory location." RDLK is thus test-and-set! Dave Dahm created spin locks ("Buzz Locks") and a two-phase lock called "Dahm Locks."*
- [M15] "OSSpinLock Is Unsafe" by J. McCall. mjtsai.com/blog/2015/12/16/osspinlock-is-unsafe. *Calling OSSpinLock on a Mac is unsafe when using threads of different priorities – you might spin forever! So be careful, Mac fanatics, even your mighty system can be less than perfect...*
- [MS91] "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors" by 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" by G.L. Peterson. Information Processing Letters, 12(3), pages 115–116, 1981. *Peterson's algorithm introduced here.*
- [R97] "What Really Happened on Mars?" by Glenn E. Reeves. [research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html](http://en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html). *A description of priority inversion on Mars Pathfinder. Concurrent code correctness matters, especially in space!*
- [S05] "Guide to porting from Solaris to Linux on x86" by Ajay Sood, April 29, 2005. Available: <http://www.ibm.com/developerworks/linux/library/l-solar/>.
- [S09] "OpenSolaris Thread Library" by Sun.. Code: src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/lib/libc/port/threads/synch.c. *Pretty interesting, although who knows what will happen now that Oracle owns Sun. Thanks to Mike Swift for the pointer.*
- [W09] "Load-Link, Store-Conditional" by Many authors.. en.wikipedia.org/wiki/Load-Link_Store-Conditional. *Can you believe we referenced Wikipedia? But, we found the information there and it felt wrong not to. Further, it was useful, listing the instructions for the different architectures: ldl.l/stl.c and ldq.l/stq.c (Alpha), lwarx/stwcx (PowerPC), ll/sc (MIPS), and ldrex/strex (ARM). Actually Wikipedia is pretty amazing, so don't be so harsh, OK?*
- [WG00] "The SPARC Architecture Manual: Version 9" by D. Weaver, T. Germond. SPARC International, 2000. <http://www.sparc.org/standards/SPARCV9.pdf>. *See developers.sun.com/solaris/articles/atomic_sparc/ for more on atomics.*

Homework (Simulation)

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 answer the questions below.

Questions

1. Examine `flag.s`. This code “implements” locking with a single memory flag. Can you understand the assembly?
2. When you run with the defaults, does `flag.s` work? 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`?
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 `p 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? Make sure to set the thread IDs appropriately (using `-a bx=0,bx=1` for example) as the code assumes it.
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? Then run with the following flags: `-a bx=1000,bx=1000` (causing each thread to loop through the critical section 1000 times). Watch what happens; do the threads spend much time spin-waiting for the lock?
12. How does the code behave as you add more threads?
13. Now examine `yield.s`, in which a `yield` instruction enables one thread to yield control of the CPU (realistically, this would be an OS primitive, but for the simplicity, we assume an instruction 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?
14. 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`?

Condition Variables

Another type of synchronization

- Locks allow one type of synchronization between threads – mutual exclusion
- Another common requirement in multi-threaded applications – waiting and signaling
 - E.g., Thread T1 wants to continue only after T2 has finished some task
- Can accomplish this by busy-waiting on some variable, but inefficient
- Need a new synchronization primitive: condition variables

Condition Variables

- A condition variable (CV) is a queue that a thread can put itself into when waiting on some condition
- Another thread that makes the condition true can signal the Cv to wake up a waiting thread
- Pthreads provides CV for user programs
 - OS has a similar functionality of wait/signal for kernel threads
- Signal wakes up one thread, signal broadcast wakes up all waiting threads

Example: parent waits for child

```
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

Why check condition in while loop?

- In the example code, why do we check condition before calling wait?
 - In case the child has already run and done is true, then no need to wait
- Why check condition with “while” loop and not “if”?
 - To avoid corner cases of thread being woken up even when condition not true (may be an issue with some implementations)

```
if(condition)
    wait(condvar)
//small chance that condition may be false when wait returns

while(condition)
    wait(condvar)
//condition guaranteed to be true since we check in while-loop
```

Why use lock when calling wait?

What if no lock is held when calling wait/signal?

```
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 }
```

- Race condition: missed wakeup
 - Parent checks done to be 0, decides to sleep, interrupted
 - Child runs, sets done to 0, signals, but no one sleeping yet
 - Parent now resumes and goes to sleep forever
- Lock must be held when calling wait and signal with CV
- The wait function releases the lock before putting thread to sleep, so lock is available for signaling thread

Example: Producer/Consumer problem

- A common pattern in multi-threaded programs
- Example: in a multi-threaded web server, one thread accepts requests from the network and puts them in a queue. Worker threads get requests from this queue and process them.
- Setup: one or more producer threads, one or more consumer threads, a shared buffer of bounded size

Producer/Consumer with 2 CVs

```
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 == MAX)
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);
```

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

```

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

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.

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

```

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

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);
```

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? (Figure 30.4)

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.

```

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 }
```

Figure 30.4: Parent Waiting: No State Variable

```

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 }
```

Figure 30.5: Parent Waiting: No Lock

Here (Figure 30.5) 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 it¹!

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.

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.

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.

¹Note that this example is not “real” code, because the call to `pthread_cond_wait()` always requires a mutex as well as a condition variable; here, we just pretend that the interface does not do so for the sake of the negative example.

```

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 30.6: The Put And Get Routines (Version 1)

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.

Because the bounded buffer is a shared resource, we must of course require synchronized access to it, lest² 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

²This is where we drop some serious Old English on you, and the subjunctive form.

```

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 30.7: Producer/Consumer Threads (Version 1)

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 30.6 (page 6) 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 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 30.7 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.

```

1 int loops; // must initialize somewhere...
2 cond_t cond;
3 mutex_t mutex;
4
5 void *producer(void *arg) {
6     int i;
7     for (i = 0; i < loops; i++) {
8         Pthread_mutex_lock(&mutex); // p1
9         if (count == 1) // p2
10             Pthread_cond_wait(&cond, &mutex); // p3
11         put(i); // p4
12         Pthread_cond_signal(&cond); // p5
13         Pthread_mutex_unlock(&mutex); // p6
14     }
15 }
16
17 void *consumer(void *arg) {
18     int i;
19     for (i = 0; i < loops; i++) {
20         Pthread_mutex_lock(&mutex); // c1
21         if (count == 0) // c2
22             Pthread_cond_wait(&cond, &mutex); // c3
23         int tmp = get(); // c4
24         Pthread_cond_signal(&cond); // c5
25         Pthread_mutex_unlock(&mutex); // c6
26         printf("%d\n", tmp);
27     }
28 }
```

Figure 30.8: Producer/Consumer: Single CV And If Statement

Not surprisingly, that something more is some condition variables. In this (broken) first try (Figure 30.8), we have a single condition variable `cond` and associated lock `mutex`.

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 30.8 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

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Run		Ready		Ready	0	
c2	Run		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Run	0	
	Sleep		Ready	p2	Run	0	
	Sleep		Ready	p4	Run	1	Buffer now full
	Ready		Ready	p5	Run	1	T_{c1} awoken
	Ready		Ready	p6	Run	1	
	Ready		Ready	p1	Run	1	
	Ready		Ready	p2	Run	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Run		Sleep	1	T_{c2} sneaks in ...
	Ready	c2	Run		Sleep	1	
	Ready	c4	Run		Sleep	0	... and grabs data
	Ready	c5	Run		Ready	0	T_p awoken
	Ready	c6	Run		Ready	0	
c4	Run		Ready		Ready	0	Oh oh! No data

Figure 30.9: Thread Trace: Broken Solution (Version 1)

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 30.9 shows the action each thread takes, as well as its scheduler state (Ready, Running, or Sleeping) over time.

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

```

1 int loops;
2 cond_t cond;
3 mutex_t mutex;
4
5 void *producer(void *arg) {
6     int i;
7     for (i = 0; i < loops; i++) {
8         Pthread_mutex_lock(&mutex);           // p1
9         while (count == 1)                   // p2
10            Pthread_cond_wait(&cond, &mutex); // p3
11        put(i);                          // p4
12        Pthread_cond_signal(&cond);       // p5
13        Pthread_mutex_unlock(&mutex);    // p6
14    }
15 }
16
17 void *consumer(void *arg) {
18     int i;
19     for (i = 0; i < loops; i++) {
20         Pthread_mutex_lock(&mutex);           // c1
21         while (count == 0)                   // c2
22            Pthread_cond_wait(&cond, &mutex); // c3
23         int tmp = get();                  // c4
24         Pthread_cond_signal(&cond);       // c5
25         Pthread_mutex_unlock(&mutex);    // c6
26         printf("%d\n", tmp);
27     }
28 }
```

Figure 30.10: Producer/Consumer: Single CV And While

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 30.10): 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.

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! (*pause for you to think, or close your eyes...*)

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Run		Ready		Ready	0	
c2	Run		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	
	Sleep	c1	Run		Ready	0	Nothing to get
	Sleep	c2	Run		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Run	0	
	Sleep		Sleep	p2	Run	0	
	Sleep		Sleep	p4	Run	1	Buffer now full
	Ready		Sleep	p5	Run	1	T_{c1} awoken
	Ready		Sleep	p6	Run	1	
	Ready		Sleep	p1	Run	1	
	Ready		Sleep	p2	Run	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Run		Sleep		Sleep	1	Recheck condition
c4	Run		Sleep		Sleep	0	T_{c1} grabs data
c5	Run		Ready		Sleep	0	Oops! Woke T_{c2}
c6	Run		Ready		Sleep	0	
c1	Run		Ready		Sleep	0	
c2	Run		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Run		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep...

Figure 30.11: Thread Trace: Broken Solution (Version 2)

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, the producer runs, puts a value in the buffer, and wakes one of the consumers (say T_{c1}). The producer then loops back (releasing and re-acquiring the lock along the way) and tries to put more data in the buffer; because the buffer is full, the producer instead waits on the condition (thus sleeping). Now, one consumer is ready to run (T_{c1}), and two threads are sleeping on a condition (T_{c2} and T_p). We are about to cause a problem: 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 *only 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

```

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 30.12: Producer/Consumer: Two CVs And While

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 30.11 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.

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 30.12 shows the resulting code.

In the code, 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.

```

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 30.13: The Correct 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 30.14: The Correct Producer/Consumer Synchronization

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.

The Correct 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.

The first change for this correct solution is within the buffer structure itself and the corresponding `put()` and `get()` (Figure 30.13). We also slightly change the conditions that producers and consumers check in order to determine whether to sleep or not. We also show the correct waiting and signaling logic (Figure 30.14). 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; time to sit back and drink a cold one.

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 30.15 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?

```

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 30.15: Covering Conditions: An Example

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.

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" by Edsger W. Dijkstra. 1968. Available online here: <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" by 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" by 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" by 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" by Mysterious author. March, 2011. Available online: http://linux.die.net/man/3/pthread_cond_signal. *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" by 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" by George Orwell. Secker and Warburg, 1949. *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 (Code)

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. Read the README for details.

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. Run with one producer and one consumer, and have the producer produce a few values. Start with a buffer (size 1), and then increase it. How does the behavior of the code change with larger buffers? (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 0,0,0,0,0,0,0,1`?
3. If possible, run the code on different systems (e.g., a Mac and Linux). Do you see different behavior across these systems?
4. Let's look at some timings. 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? `./main-two-cvs-while -p 1 -c 3 -m 1 -C 0,0,0,1,0,0,0:0,0,0,1,0,0,0,0:0,0,0,1,0,0,0 -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), again using a single-entry buffer. What time do you predict in this case? `./main-two-cvs-while -p 1 -c 3 -m 1 -C 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

What is a semaphore?

- Synchronization primitive like condition variables
- Semaphore is a variable with an underlying counter
- Two functions on a semaphore variable
 - Up/post increments the counter
 - Down/wait decrements the counter and blocks the calling thread if the resulting value is negative
- A semaphore with init value 1 acts as a simple lock (binary semaphore = mutex)

```
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)

Semaphores for ordering

- Can be used to set order of execution between threads like CV
- Example: parent waiting for child (init = 0)

```
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 }
```

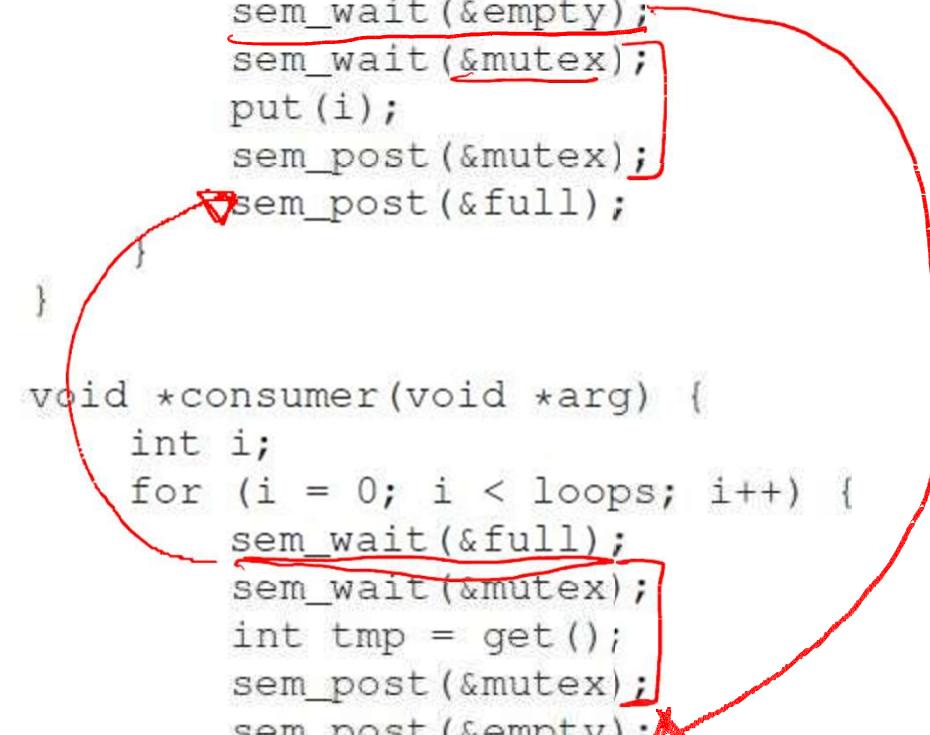
Example: Producer/Consumer (1)

- Need two semaphores for signaling
 - One to track empty slots, and make producer wait if no more empty slots
 - One to track full slots, and make consumer wait if no more full slots
- One semaphore to act as mutex for buffer

```
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 }
```

Example: Producer/Consumer (2)

```
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);
9            sem_wait(&mutex);
10       put(i);
11       sem_post(&mutex);
12       sem_post(&full);
13    }
14 }
15
16 void *consumer(void *arg) {
17    int i;
18    for (i = 0; i < loops; i++) {
19        sem_wait(&full);
20        sem_wait(&mutex);
21        int tmp = get();
22        sem_post(&mutex);
23        sem_post(&empty);
24        printf("%d\n", tmp);
25    }
26 }
```



Incorrect solution with deadlock

- What if lock is acquired before signaling?
- Waiting thread sleeps with mutex and the signaling thread can never wake it up

```
5 void *producer(void *arg) {
6     int i;
7     for (i = 0; i < loops; i++) {
8         sem_wait(&mutex);
9         sem_wait(&empty);
10        put(i);
11        sem_post(&full);
12        sem_post(&mutex);
13    }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&mutex);
20         sem_wait(&full);
21         int tmp = get();
22         sem_post(&empty);
23         sem_post(&mutex);
24         printf("%d\n", tmp);
25     }
26 }
```

Semaphores

As we know now, one needs both locks and condition variables to solve a broad range of relevant and interesting concurrency problems. One of the first people to realize this years ago was **Edsger Dijkstra** (though it is hard to know the exact history [GR92]), known among other things for his famous “shortest paths” algorithm in graph theory [D59], an early polemic on structured programming entitled “Goto Statements Considered Harmful” [D68a] (what a great title!), and, in the case we will study here, the introduction of a synchronization primitive called the **semaphore** [D68b, D72]. Indeed, Dijkstra and colleagues invented the semaphore as a single primitive for all things related to synchronization; as you will see, one can use semaphores as both locks and condition variables.

THE CRUX: HOW TO USE SEMAPHORES

How can we use semaphores instead of locks and condition variables? What is the definition of a semaphore? What is a binary semaphore? Is it straightforward to build a semaphore out of locks and condition variables? To build locks and condition variables out of semaphores?

31.1 Semaphores: A Definition

A semaphore is an object with an integer value that we can manipulate with two routines; in the POSIX standard, these routines are `sem_wait()` and `sem_post()`¹. Because the initial value of the semaphore determines its behavior, before calling any other routine to interact with the semaphore, we must first initialize it to some value, as the code in Figure 31.1 does.

¹Historically, `sem_wait()` was called `P()` by Dijkstra and `sem_post()` called `V()`. These shortened forms come from Dutch words; interestingly, which Dutch words they supposedly derive from has changed over time. Originally, `P()` came from “passering” (to pass) and `V()` from “vrijgave” (release); later, Dijkstra wrote `P()` was from “prolaag”, a contraction of “probeer” (Dutch for “try”) and “verlaag” (“decrease”), and `V()` from “verhoog” which means “increase”. Sometimes, people call them down and up. Use the Dutch versions to impress your friends, or confuse them, or both. See <https://news.ycombinator.com/item?id=876> for details.

```

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.

```

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 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)

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.

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

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

Val	Thread 0	State	Thread 1	State
1		Run		Ready
1	call sem_wait()	Run		Ready
0	sem_wait() returns	Run		Ready
0	(crit sect begin)	Run		Ready
0	<i>Interrupt; Switch→T1</i>	Ready		Run
0		Ready	call sem_wait()	Run
-1		Ready	decr sem	Run
-1		Ready	(sem<0) → sleep	Sleep
-1		Run	<i>Switch→T0</i>	Sleep
-1	(crit sect end)	Run		Sleep
-1	call sem_post()	Run		Sleep
0	incr sem	Run		Sleep
0	wake(T1)	Run		Ready
0	sem_post() returns	Run		Ready
0	<i>Interrupt; Switch→T1</i>	Ready		Run
0		Ready	sem_wait() returns	Run
0		Ready	(crit sect)	Run
0		Ready	call sem_post()	Run
1		Ready	sem_post() returns	Run

Figure 31.5: Thread Trace: Two Threads Using A Semaphore

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.

Figure 31.5 shows a trace of this example. In addition to thread actions, the figure shows the **scheduler state** of each thread: Run (the thread is running), Ready (i.e., runnable but not running), and Sleep (the thread is blocked). Note 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 For Ordering

Semaphores are also useful to order events in a concurrent program. For example, a thread may wish to wait for a list to become non-empty,

```

1 sem_t s;
2
3 void *child(void *arg) {
4     printf("child\n");
5     sem_post(&s); // signal here: child is done
6     return NULL;
7 }
8
9 int main(int argc, char *argv[]) {
10    sem_init(&s, 0, X); // what should X be?
11    printf("parent: begin\n");
12    pthread_t c;
13    Pthread_create(&c, NULL, child, NULL);
14    sem_wait(&s); // wait here for child
15    printf("parent: end\n");
16    return 0;
17 }
```

Figure 31.6: A Parent Waiting For Its Child

so it can delete an element from it. In this pattern of usage, we often find one thread *waiting* for something to happen, and another thread making that something happen and then *signaling* that it has happened, thus waking the waiting thread. We are thus using the semaphore as an **ordering** primitive (similar to our use of **condition variables** earlier).

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

Val	Parent	State	Child	State
0	create(Child)	Run	(Child exists, can run)	Ready
0	call sem_wait()	Run		Ready
-1	decr sem	Run		Ready
-1	$(sem < 0) \rightarrow sleep$	Sleep		Ready
-1	Switch→Child	Sleep	child runs	Run
-1		Sleep	call sem_post()	Run
0		Sleep	inc sem	Run
0		Ready	wake (Parent)	Run
0		Ready	sem_post() returns	Run
0		Ready	Interrupt→Parent	Ready
0	sem_wait() returns	Run		Ready

Figure 31.7: Thread Trace: Parent Waiting For Child (Case 1)

Val	Parent	State	Child	State
0	create(Child)	Run	(Child exists; can run)	Ready
0	Interrupt→Child	Ready	child runs	Run
0		Ready	call sem_post()	Run
1		Ready	inc sem	Run
1		Ready	wake (nobody)	Run
1		Ready	sem_post() returns	Run
1	parent runs	Run	Interrupt→Parent	Ready
1	call sem_wait()	Run		Ready
0	decrement sem	Run		Ready
0	$(sem \geq 0) \rightarrow awake$	Run		Ready
0	sem_wait() returns	Run		Ready

Figure 31.8: Thread Trace: Parent Waiting For Child (Case 2)

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) 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.

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.

ASIDE: SETTING THE VALUE OF A SEMAPHORE

We've now seen two examples of initializing a semaphore. In the first case, we set the value to 1 to use the semaphore as a lock; in the second, to 0, to use the semaphore for ordering. So what's the general rule for semaphore initialization?

One simple way to think about it, thanks to Perry Kivolowitz, is to consider the number of resources you are willing to give away immediately after initialization. With the lock, it was 1, because you are willing to have the lock locked (given away) immediately after initialization. With the ordering case, it was 0, because there is nothing to give away at the start; only when the child thread is done is the resource created, at which point, the value is incremented to 1. Try this line of thinking on future semaphore problems, and see if it helps.

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 (page 8).

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 Figure 31.10, 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 are empty
26     sem_init(&full, 0, 0);   // 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 return from `sem_wait(&full)` (Line C1), find that the buffer was full, and 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.

```

1 void *producer(void *arg) {
2     int i;
3     for (i = 0; i < loops; i++) {
4         sem_wait(&mutex);           // Line P0 (NEW LINE)
5         sem_wait(&empty);          // Line P1
6         put(i);                  // Line P2
7         sem_post(&full);          // Line P3
8         sem_post(&mutex);          // Line P4 (NEW LINE)
9     }
10 }
11
12 void *consumer(void *arg) {
13     int i;
14     for (i = 0; i < loops; i++) {
15         sem_wait(&mutex);           // Line C0 (NEW LINE)
16         sem_wait(&full);           // Line C1
17         int tmp = get();           // Line C2
18         sem_post(&empty);          // Line C3
19         sem_post(&mutex);          // Line C4 (NEW LINE)
20         printf("%d\n", tmp);
21     }
22 }
```

Figure 31.11: Adding Mutual Exclusion (Incorrectly)

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 action 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?

```

1 void *producer(void *arg) {
2     int i;
3     for (i = 0; i < loops; i++) {
4         sem_wait(&empty);           // Line P1
5         sem_wait(&mutex);          // Line P1.5 (MUTEX HERE)
6         put(i);                  // Line P2
7         sem_post(&mutex);          // Line P2.5 (AND HERE)
8         sem_post(&full);           // Line P3
9     }
10 }
11
12 void *consumer(void *arg) {
13     int i;
14     for (i = 0; i < loops; i++) {
15         sem_wait(&full);          // Line C1
16         sem_wait(&mutex);          // Line C1.5 (MUTEX HERE)
17         int tmp = get();           // Line C2
18         sem_post(&mutex);          // Line C2.5 (AND HERE)
19         sem_post(&empty);           // Line C3
20         printf("%d\n", tmp);
21     }
22 }
```

Figure 31.12: Adding Mutual Exclusion (Correctly)

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.

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.

At Last, A Working Solution

To solve this problem, we simply must reduce the scope of the lock. Figure 31.12 (page 10) shows the correct 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 outside². 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, or during a job interview.

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 (page 12).

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 acquire 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 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, especially 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.

²Indeed, it may have been more natural to place the `mutex` acquire/release inside the `put()` and `get()` functions for the purposes of modularity.

```

1  typedef struct _rwlock_t {
2      sem_t lock;          // binary semaphore (basic lock)
3      sem_t writelock;    // allow ONE writer/MANY readers
4      int   readers;      // #readers 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) // first reader gets writelock
17         sem_wait(&rw->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) // last reader lets it go
25         sem_post(&rw->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

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.

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**.

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.

Here is the basic loop of each philosopher, assuming each has a unique thread identifier p from 0 to 4 (inclusive):

```
while (1) {
    think();
    get_forks(p);
    eat();
    put_forks(p);
}
```

The key challenge, then, is to write the routines `get_forks()` and `put_forks()` such that there is no deadlock, no philosopher starves and

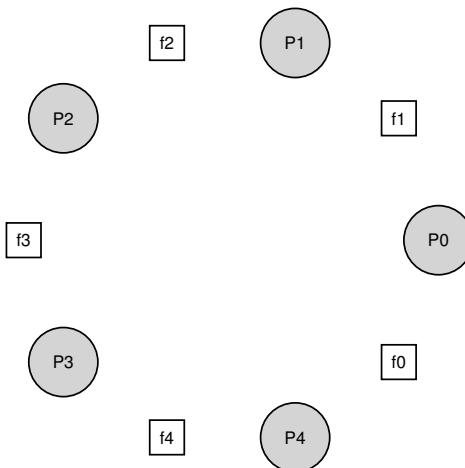


Figure 31.14: The Dining Philosophers

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]`.

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 `get_forks()` and `put_forks()` routine (Figure 31.15, page 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,

```

1 void get_forks(int p) {
2     sem_wait(&forks[left(p)]);
3     sem_wait(&forks[right(p)]);
4 }
5
6 void put_forks(int p) {
7     sem_post(&forks[left(p)]);
8     sem_post(&forks[right(p)]);
9 }
```

Figure 31.15: The `get_forks()` And `put_forks()` Routines

```

1 void get_forks(int p) {
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 }
```

Figure 31.16: Breaking The Dependency In `get_forks()`

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) gets the forks in a *different* order than the others (Figure 31.16); the `put_forks()` code remains the same.

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.

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 Thread Throttling

One other simple use case for semaphores arises on occasion, and thus we present it here. The specific problem is this: how can a programmer prevent “too many” threads from doing something at once and bogging the system down? Answer: decide upon a threshold for “too many”, and then use a semaphore to limit the number of threads concurrently executing the piece of code in question. We call this approach **throttling** [T99], and consider it a form of **admission control**.

Let’s consider a more specific example. Imagine that you create hundreds of threads to work on some problem in parallel. However, in a certain part of the code, each thread acquires a large amount of memory to perform part of the computation; let’s call this part of the code the *memory-intensive region*. If *all* of the threads enter the memory-intensive region at the same time, the sum of all the memory allocation requests will exceed the amount of physical memory on the machine. As a result, the machine will start thrashing (i.e., swapping pages to and from the disk), and the entire computation will slow to a crawl.

A simple semaphore can solve this problem. By initializing the value of the semaphore to the maximum number of threads you wish to enter the memory-intensive region at once, and then putting a `sem_wait()` and `sem_post()` around the region, a semaphore can naturally throttle the number of threads that are ever concurrently in the dangerous region of the code.

31.8 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 31.17 (page 17).

In the code above, 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.

```
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 31.17: Implementing Zemaphores With Locks And CVs

Curiously, building condition variables out of semaphores 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 of a problem than it might appear.

31.9 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 understand-

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.

ing 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" by 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.*
- [C08] "Real-world Concurrency" by Bryan Cantrill, 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" by 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" by 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" by E.W. Dijkstra. CACM, volume 11(3), March 1968. <http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF>. *Sometimes thought of as the beginning of the field of software engineering.*
- [D68b] "The Structure of the THE Multiprogramming System" by E.W. Dijkstra. CACM, volume 11(5), 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" by E.W. Dijkstra. Information Processing Letters 1, 1972. <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, practitioners in OS design knew of many of the problems described by Dijkstra, so perhaps giving him too much credit would be a misrepresentation.*
- [D08] "The Little Book of Semaphores" by A.B. Downey. Available at the following site: <http://greenteapress.com/semaphores/>. *A nice (and free!) book about semaphores. Lots of fun problems to solve, if you like that sort of thing.*
- [D71] "Hierarchical ordering of sequential processes" by E.W. Dijkstra. Available online here: <http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF>. *Presents numerous concurrency problems, including Dining Philosophers. The wikipedia page about this problem is also useful.*
- [GR92] "Transaction Processing: Concepts and Techniques" by Jim Gray, 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." Oh, snap!*
- [H87] "Aspects of Cache Memory and Instruction Buffer Performance" by 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" by 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.*
- [T99] "Re: NT kernel guy playing with Linux" by Linus Torvalds. June 27, 1999. Available: <https://yarchive.net/comp/linux/semaphores.html>. *A response from Linus himself about the utility of semaphores, including the throttling case we mention in the text. As always, Linus is slightly insulting but quite informative.*

Homework (Code)

In this homework, we'll use semaphores to solve some well-known concurrency problems. Many of these are taken from Downey's excellent "Little Book of Semaphores"³, which does a good job of pulling together a number of classic problems as well as introducing a few new variants; interested readers should check out the Little Book for more fun.

Each of the following questions provides a code skeleton; your job is to fill in the code to make it work given semaphores. On Linux, you will be using native semaphores; on a Mac (where there is no semaphore support), you'll have to first build an implementation (using locks and condition variables, as described in the chapter). Good luck!

Questions

1. The first problem is just to implement and test a solution to the **fork/join problem**, as described in the text. Even though this solution is described in the text, the act of typing it in on your own is worthwhile; even Bach would rewrite Vivaldi, allowing one soon-to-be master to learn from an existing one. See `fork-join.c` for details. Add the call `sleep(1)` to the child to ensure it is working.
2. Let's now generalize this a bit by investigating the **rendezvous problem**. The problem is as follows: you have two threads, each of which are about to enter the rendezvous point in the code. Neither should exit this part of the code before the other enters it. Consider using two semaphores for this task, and see `rendezvous.c` for details.
3. Now go one step further by implementing a general solution to **barrier synchronization**. Assume there are two points in a sequential piece of code, called P_1 and P_2 . Putting a **barrier** between P_1 and P_2 guarantees that all threads will execute P_1 before any one thread executes P_2 . Your task: write the code to implement a `barrier()` function that can be used in this manner. It is safe to assume you know N (the total number of threads in the running program) and that all N threads will try to enter the barrier. Again, you should likely use two semaphores to achieve the solution, and some other integers to count things. See `barrier.c` for details.
4. Now let's solve the **reader-writer problem**, also as described in the text. In this first take, don't worry about starvation. See the code in `reader-writer.c` for details. Add `sleep()` calls to your code to demonstrate it works as you expect. Can you show the existence of the starvation problem?
5. Let's look at the reader-writer problem again, but this time, worry about starvation. How can you ensure that all readers and writers eventually make progress? See `reader-writer-nostarve.c` for details.
6. Use semaphores to build a **no-starve mutex**, in which any thread that tries to acquire the mutex will eventually obtain it. See the code in `mutex-nostarve.c` for more information.
7. Liked these problems? See Downey's free text for more just like them. And don't forget, have fun! But, you always do when you write code, no?

³Available: <http://greenteapress.com/semaphores/downey08semaphores.pdf>.

Reader-writer

Writer

Wait(mutex)

write here

Signal(wrmutex)

Reader

wait(mutex)

readers++

if(reader == 1)

wait(wrmutex)

Signal(mutex)

read here

wait(mutex)

readers--

if(readers == 0)

Signal(wrmutex)

Signal(mutex)

first scenario :

- multiple writer want to write to same file.

- Semaphore wrmutex = 1

writer 1

wait(wrmutex)

write here

signal(wrmutex)

writer 2

wait(wrmutex)

write here

signal(wrmutex)

Second Scenario: $w\lambda mutex = 1$

• Two Semaphores $mutex = 1$

variable $readers = 0$

Writer 1

$wait(mutex)$

write here

$signal(mutex,$

Reader

$wait(mutex)$

$readers++$

$if(readers == 1)$

$wait(mutex)$

$Signal(mutex)$

Read here

$wait(mutex)$

$readers--$

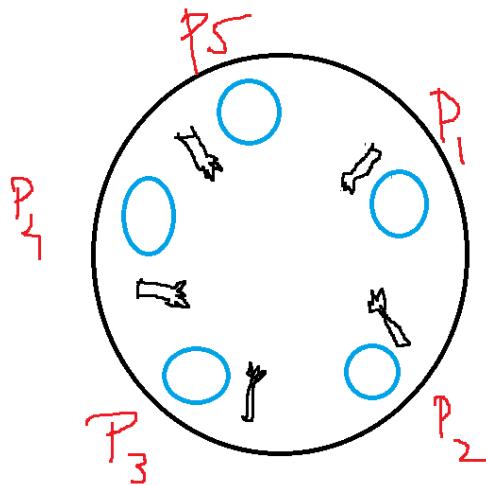
$if(readers == 0)$

$Signal(mutex)$

$signal(mutex)$

DINNING PHILOSPHER PROBLEM

Let say that we have five philosopher P1, P2, P3, P4, P5 who are sitting in the table.



Infront of them there are five plates for each of the philosophers.

Five forks

Each philosopher can do two things eat (or) think

Condition to eat is;

Philosopher need to have both the forks (left forks and right forks)

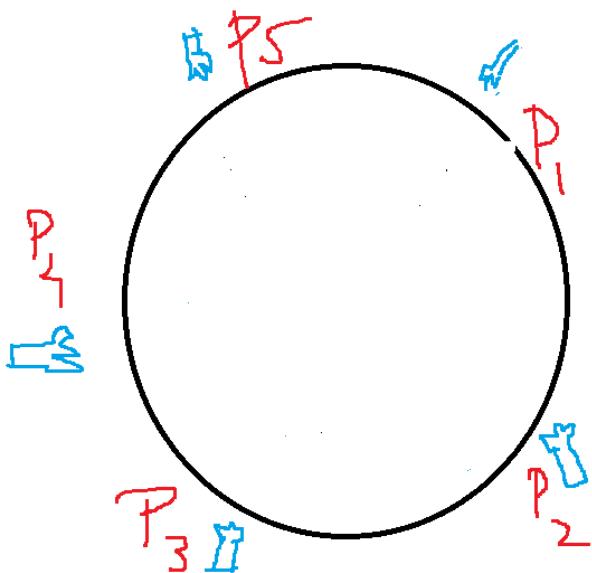
Problem:

Develop an algorithm where no philosopher starves.

First solution (NAÏVE)

```
#define N  
Void philosopher (int i)  
{  
    While (TRUE)  
{  
        Think ();  
        Take_fork (Ri);  
        Take_fork (Li);  
        Eat ();  
        Put_fork (Li);  
        Put_fork (Ri);  
    }  
}
```

Possible issues



Leads to Deadlock

Second solution:

```
#define N  
Void Philosopher (int i)  
{  
    While (TRUE)  
    {  
        Think ();  
        Take_fork (Ri);  
        If available (Li)  
        {  
            Take_fork (Li)  
            Eat ();  
            Put_fork (Ri);  
            Put_fork (Li);  
        }  
        Else  
        {  
            Put_fork (Ri);  
            Sleep (T)  
        }  
    }  
}
```

Solution using MUTEX

```
#define N  
Void Philosopher (int i)  
{  
    While (TRUE)  
    {  
        Think ();  
        Lock (mutex);  
        Take_fork (Ri);  
        Take_fork (Li);  
        Eat ();  
        Put_fork (Li);  
        Put_fork (Ri);  
        Unlock (mutex);  
    }  
}
```

Fourth solution:

Using semaphores

Semaphores S[1], S[2], S[3],.....S[N] all initialized to 0

```

void philosopher (int i)
{
    While (TRUE)
    {
        Think ();
        Take_forks(i);
        Eat ();
        Put_forks();
    }
}

```

```

Void test (int i)
{
    If (state [i] = HUNGRY && state [LEFT] != EATING && state [RIGHT] != EATING)
    {
        State [i] = EATING;
        UP(s[i]);
    }
}

```

```

Void take_forks (int i)
{
    Lock (mutex);
    State [i] = HUNGRY;
    Test (i);
    Unlock (mutex);
    Down (S[i]);
}

```

For eg:

	P1	P2	P3	P4	P5
State	T	T	T	T	T
Semaphore	0	0	0	0	0

P3 -----H

P3 ----- T-H-E

P4 -----T-H

```

Void put_forks (int i)
{
    Lock (mutex);
    State [i] = THINKING;
    Test (LEFT);
    Test (RIGHT);
    Unlock (mutex);
}

```

Concurrency Bugs

Bugs in concurrent programs

- Writing multi-threaded programs is tricky
- Bugs are non-deterministic and occur based on execution order of threads – very hard to debug
- Two types of bugs
 - Deadlocks: threads cannot execute any further and wait for each other
 - Non-deadlock bugs: non deadlock but incorrect results when threads execute

Non deadlock bugs

- Atomicity bugs – atomicity assumptions made by programmer are violated during execution of concurrent threads
 - Fix: locks for mutual exclusion
- Order-violation bugs – desired order of memory accesses is flipped during concurrent execution
 - Fix: condition variables

Atomicity bug: example

- One thread reads and prints a shared data item, while another concurrently modifies 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;
```

- Atomicity bugs can occur, not just when writing to shared data, but even when reading it

Atomicity bug example: fix

- Always use locks when accessing shared data

```
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 bug: example

- Thread1 assumes Thread2 has already run

```
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 }
```

- No assumptions can be made on order of execution of concurrent threads

Ordering violation bug example: fix

- Use condition variables or semaphores

```
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 }
```

Deadlock bugs

- Classic example: Thread1 holds lock L1 and is waiting for lock L2. Thread2 holds L2 and is waiting for L1.

Thread 1:

```
pthread_mutex_lock(L1);  
pthread_mutex_lock(L2);
```

Thread 2:

```
pthread_mutex_lock(L2);  
pthread_mutex_lock(L1);
```

- Deadlock need not always occur. Only occurs if executions overlap and context switch from a thread after acquiring only one lock.

Deadlock: a visual representation

- Cycle in a dependency graph

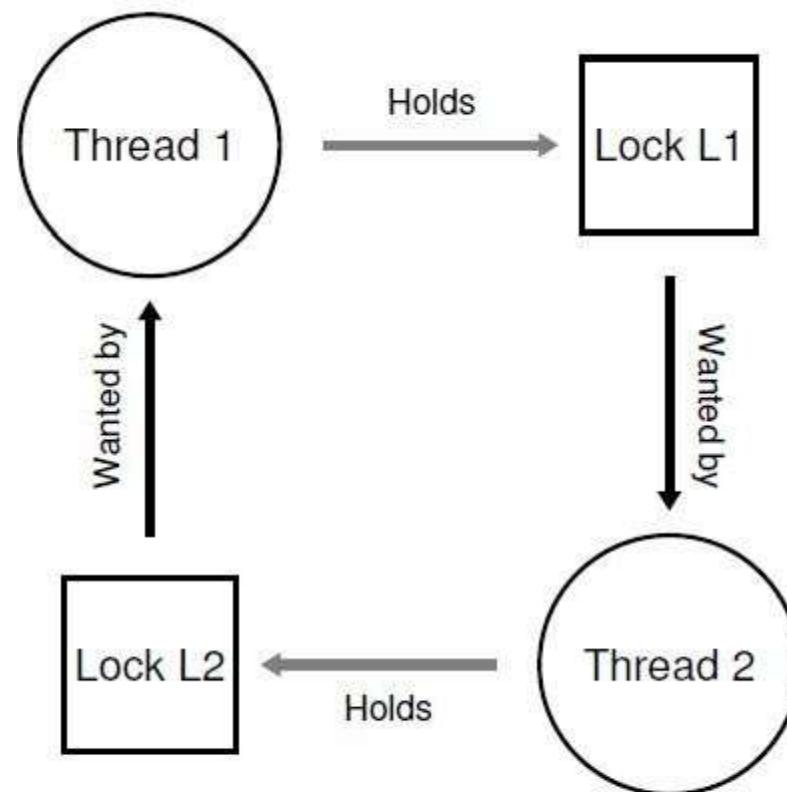


Figure 32.2: The Deadlock Dependency Graph

Conditions for deadlock

- Mutual exclusion: a thread claims exclusive control of a resource (e.g., lock)
- Hold-and-wait: thread holds a resource and is waiting for another
- No preemption: thread cannot be made to give up its resource (e.g., cannot take back a lock)
- Circular wait: there exists a cycle in the resource dependency graph
- ALL four of the above conditions must hold for a deadlock to occur

Preventing circular wait

- Acquire locks in a certain fixed order
 - E.g., both threads acquire L1 before L2
- Total ordering (or even a partial ordering on related locks) must be followed
 - E.g., order locks by address of lock variable

```
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)
```

Preventing hold-and-wait

- Acquire all locks at once, say, by acquiring a master lock first
- But this method may reduce concurrent execution and performance gains

```
1     pthread_mutex_lock(prevention);    // begin lock acquistion  
2     pthread_mutex_lock(L1);  
3     pthread_mutex_lock(L2);  
4     ...  
5     pthread_mutex_unlock(prevention); // end
```

Other solutions to deadlocks

- Deadlock avoidance: if OS knew which process needs which locks, it can schedule the processes in that deadlock will not occur
 - Banker's algorithm is very popular, but impractical in real life to assume this knowledge
 - Example, below are locks needed by threads and a possible schedule decided by OS

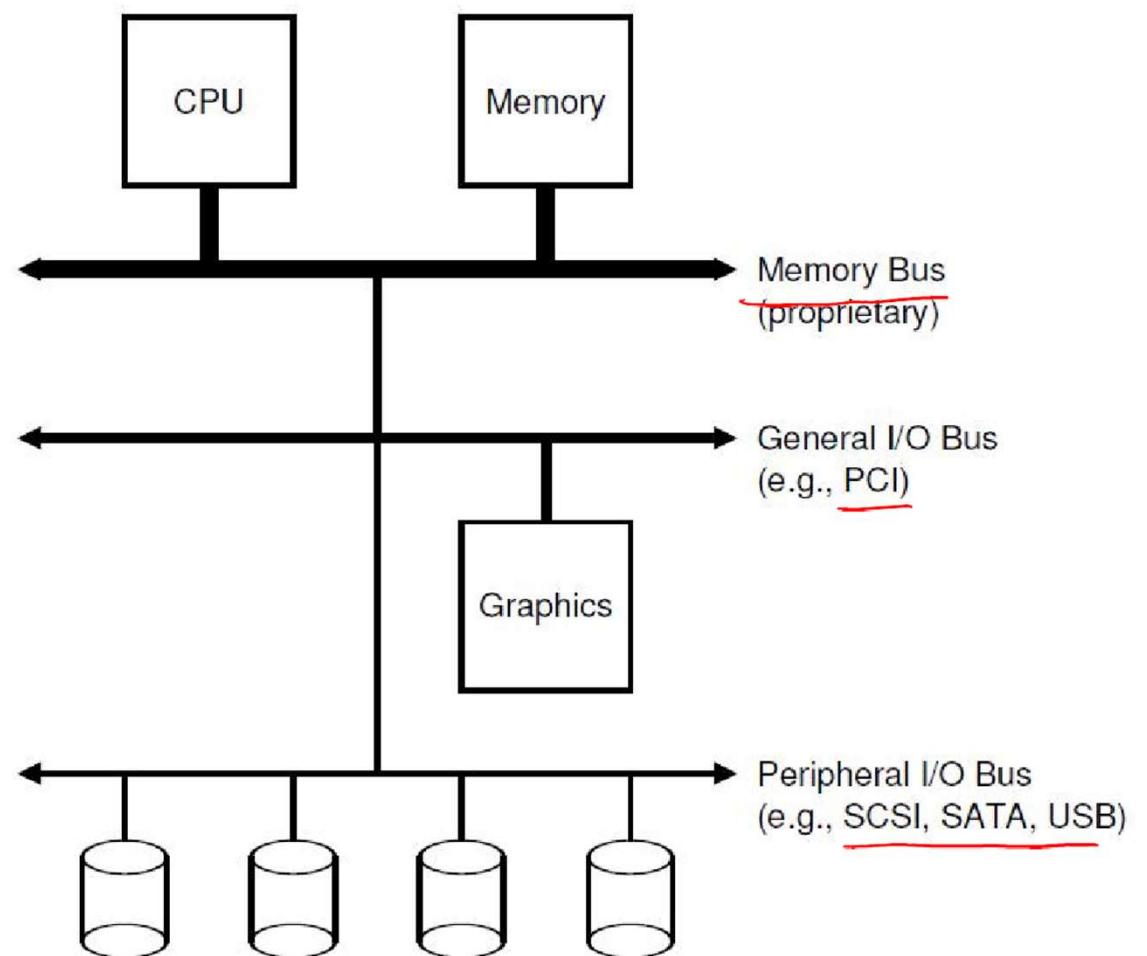


- Detect and recover: reboot system or kill deadlocked processes

Communication with I/O Devices

Input/Output Devices

- I/O devices connect to the CPU and memory via a bus
 - High speed bus, e.g., PCI
 - Other: SCSI, USB, SATA
- Point of connection to the system: port



Simple Device Model

- Block devices store a set of numbered blocks (disks)
- Character devices produce/consume stream of bytes (keyboard)
- Devices expose an interface of memory registers
 - Current status of device
 - Command to execute
 - Data to transfer
- The internals of device are usually hidden

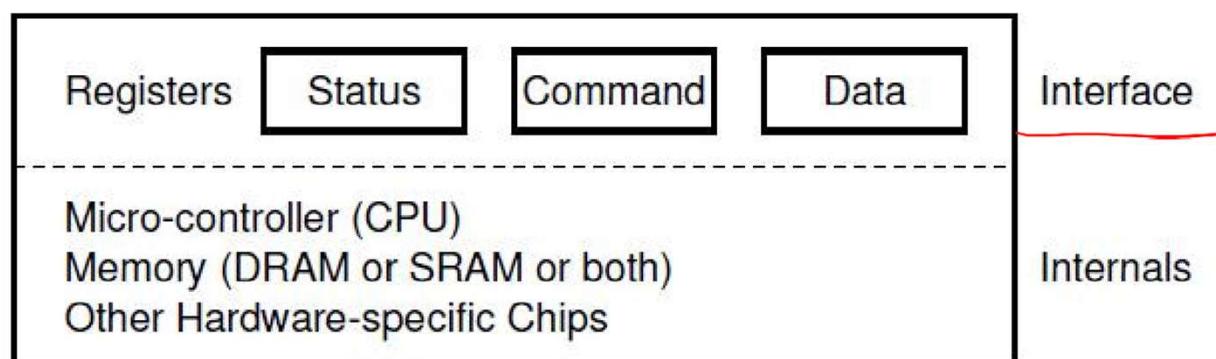


Figure 36.2: A Canonical Device

How does OS read/write to registers?

- How does OS read/write to registers like status and command?
- Explicit I/O instructions
 - E.g., on x86, `in` and `out` instructions can be used to read and write to specific registers on a device
 - Privileged instructions accessed by OS
- Memory mapped I/O
 - Device makes registers appear like memory locations
 - OS simply reads and writes from memory
 - Memory hardware routes accesses to these special memory addresses to devices

A simple execution of I/O requests

```
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
```

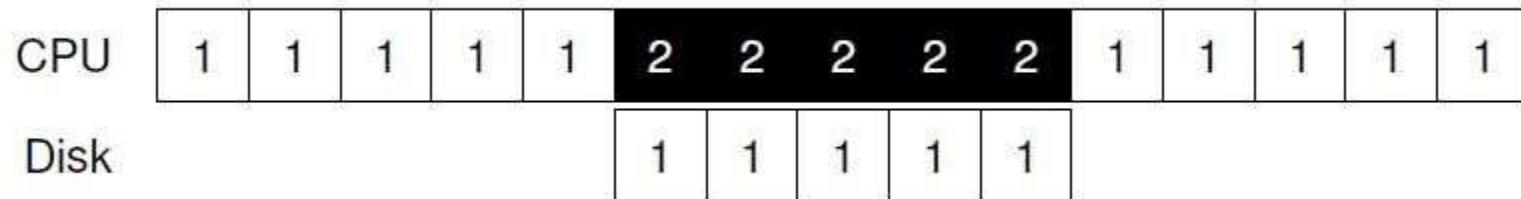
- Polling status to see if device ready – wastes CPU cycles
- Programmed I/O – CPU explicitly copies data to/from device

Interrupts

- Polling wastes CPU cycles



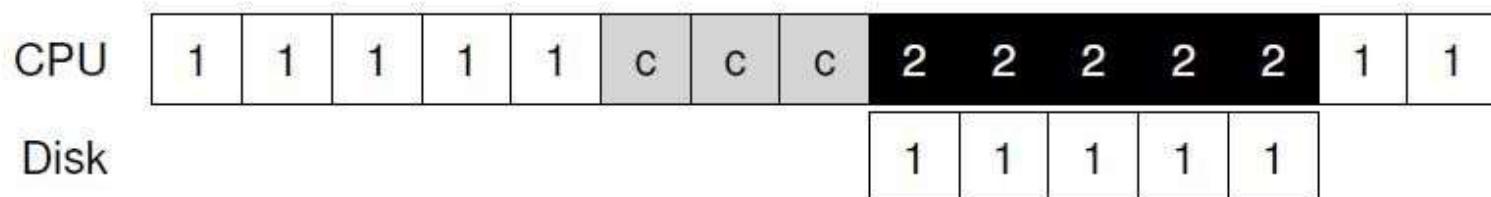
- Instead, OS can put process to sleep and switch to another process



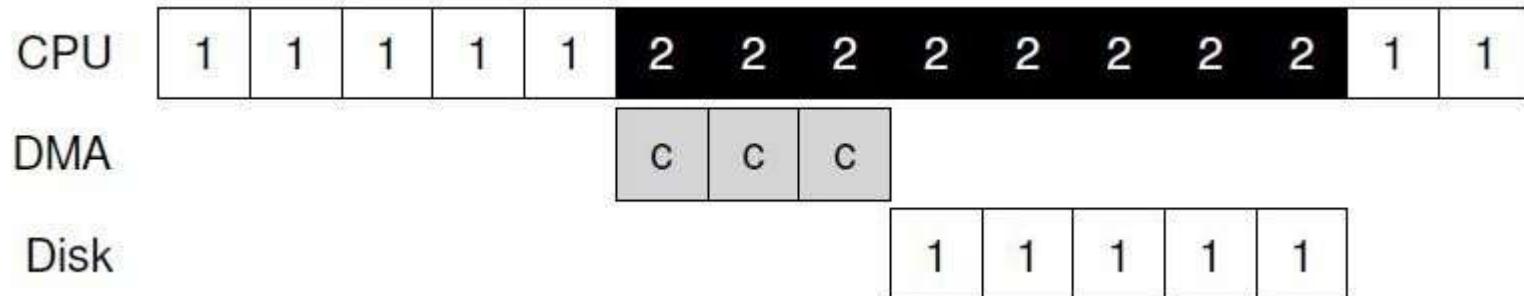
- When I/O request completes, device raises interrupt

Direct Memory Access (DMA)

- CPU cycles wasted in copying data to/from device

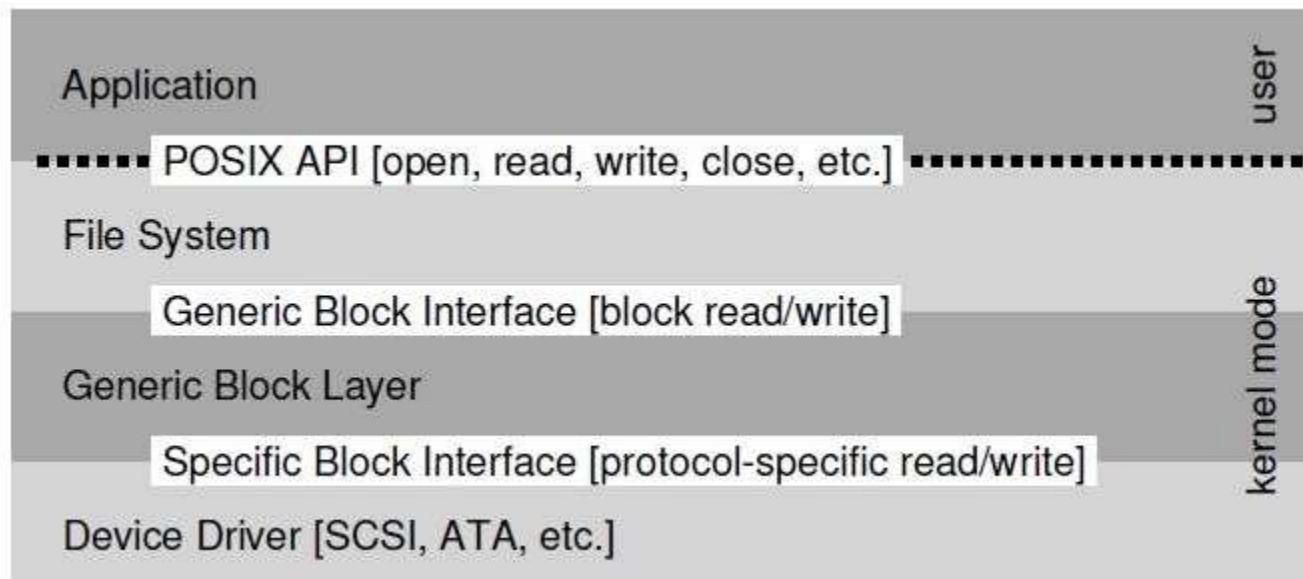


- Instead, a special piece of hardware (DMA engine) copies from main memory to device
 - CPU gives DMA engine the memory location of data
 - In case of read, interrupt raised after DMA completes
 - In case of write, disk starts writing after DMA completes



Device Driver

- Device driver: part of OS code that talks to specific device, gives commands, handles interrupts etc.
- Most OS code abstracts the device details
 - E.g., file system code is written on top of a generic block interface



Files and Directories

The file abstraction

- File – linear array of bytes, stored persistently
 - Identified with file name (human readable) and a OS-level identifier (“inode number”)
 - Inode number unique within a file system
- Directory contains other subdirectories and files, along with their inode numbers
 - Stored like a file, whose contents are filename-to-inode mappings

Directory tree

- Files and directories arranged in a tree, starting with root (“/”)

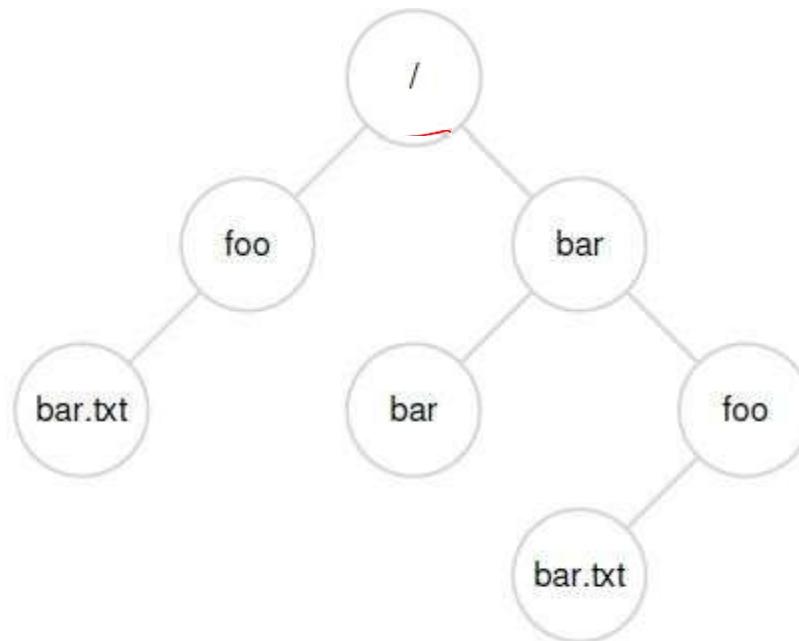


Figure 39.1: An Example Directory Tree

Operations on files (1)

- Creating a file
 - open () system call with flag to create
 - Returns a number called “file descriptor”

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
```

- Opening a file
 - Existing files must be opened before they can be read/written
 - Also uses open system call, and returns fd
- All other operations on files use the file descriptor
- close () system call closes the file

Operations on files (2)

- Reading/writing files: `read()` / `write()` system calls
 - Arguments: file descriptor, buffer with data, size
- Reading and writing happens sequentially by default
 - Successive read/write calls fetch from current offset
- What if you want to read/write at random location
 - `lseek()` system call lets you seek to random offset
- Writes are buffered in memory temporarily, so `fsync()` system call flushes all writes to disk
- Other operations: rename file, delete (unlink) file, get statistics of a file

Operations on directories

- Directories can also be accessed like files
 - Operations like create, open, read, close
- For example, the “ls” program opens and reads all directory entries
 - Directory entry contains file name, inode number, type of file (file/directory) etc.

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) {
        printf("%lu %s\n", (unsigned long) d->d_ino, d->d_name);
    }
    closedir(dp);
    return 0;
}
```

Hard links

- Hard linking creates another file that points to the same inode number (and hence, same underlying data)
- If one file deleted, file data can be accessed through the other links
- Inode maintains a link count, file data deleted only when no further links to it
- You can only unlink, OS decides when to delete

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
```

```
prompt> ls -i file file2
67158084 file
67158084 file2
prompt>
```

```
prompt> rm file
removed 'file'
prompt> cat file2
hello
```

Soft links or symbolic links

- Soft link is a file that simply stores a pointer to another filename

```
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
```

- If the main file is deleted, then the link points to an invalid entry: 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
```

Mounting a filesystem

- Mounting a filesystem connects the files to a specific point in the directory tree

```
prompt> mount -t ext3 /dev/sdal /home/users
prompt> ls /home/users/
a b
```

- Several devices and file systems are mounted on a typical machine, accessed with mount command

```
/dev/sdal 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)
```

Memory mapping a file

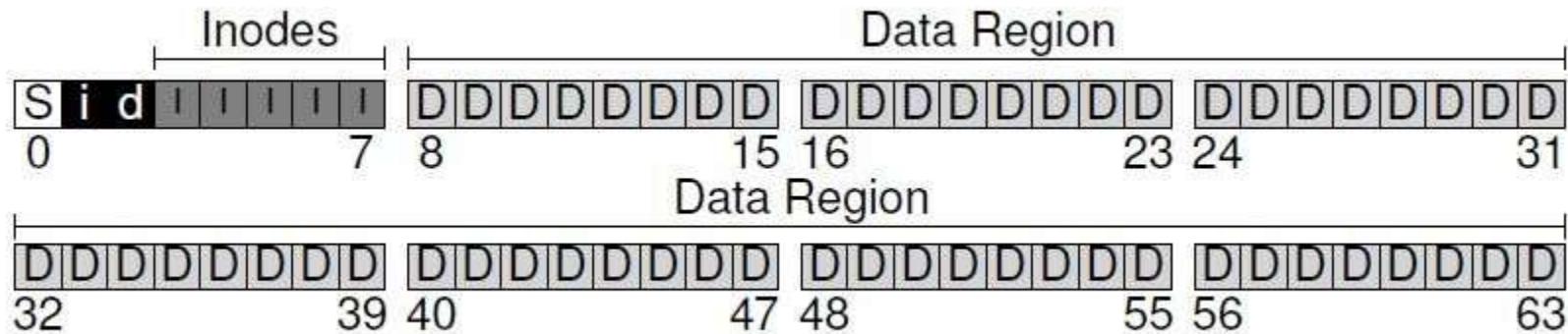
- Alternate way of accessing a file, instead of using file descriptors and read/write syscalls
- `mmap()` allocates a page in the virtual address space of a process
 - “Anonymous” page: used to store program data
 - File-backed page: contains data of file (filename provided as arg to `mmap`)
- When file is mmaped, file data copied into one or more pages in memory, can be accessed like any other memory location in program

File System Implementation

File System

- An organization of files and directories on disk
- OS has one or more file systems
- Two main aspects of file systems
 - Data structures to organize data and metadata on disk
 - Implementation of system calls like open, read, write using the data structures
- Disks expose a set of blocks (usually 512 bytes)
- File system organizes files onto blocks
 - System calls translated into reads and writes on blocks

Example: a simple file system



- Data blocks: file data stored in one or more blocks
- Metadata about every file stored in inode
 - Location of data blocks of a file, permissions etc.
- Inode blocks: each block has one or more inodes
- Bitmaps: indicate which inodes/data blocks are free
- Superblock: holds master plan of all other blocks (which are inodes, which are data blocks etc.)

Inode table

- Usually, inodes (index nodes) stored in array
 - Inode number of a file is index into this array

		The Inode Table (Closeup)																			
		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

- What does inode store?
 - File metadata: permissions, access time, etc.
 - Pointers (disk block numbers) of file data

Inode structure

- File data not stored contiguously on disk, need to track multiple block numbers of a file
- How does inode track disk block numbers?
 - Direct pointers: numbers of first few blocks are stored in inode itself (suffices for small files)
 - Indirect block: for larger files, inode stores number of indirect block, which has block numbers of file data
 - Similarly, double and triple indirect blocks (multi-level index)

File Allocation Table (FAT)

- Alternate way to track file blocks
- FAT stores next block pointer for each block
 - FAT has one entry per disk block
 - Entry has number of next file block, or null (if last block)
 - Pointer to first block stored in inode

Directory structure

- Directory stores records mapping filename to inode number, e.g., as shown below

inum	reclen	<u>strlen</u>	<u>name</u>
5	12	2	.
2	12	3	..
12	12	4	foo
13	12	4	<u>bar</u>
24	36	28	foobar_is_a.pretty_longname

- Linked list of records, or more complex structures (hash tables, binary search trees etc.)
- Directory is a special type of file and has inode and data blocks (which store the file records)

Free space management

- How to track free blocks?
 - Bitmaps, for inodes and data blocks, store one bit per block to indicate if free or not
 - Free list, super block stores pointer to first free block, a free block stores address of next block on list
 - More complex structures can also be used

Opening a file

- Why open? To have the inode readily available (in memory) for future operations on file
 - Open returns fd which points to in-memory inode
 - Reads and writes can access file data from inode
- What happens during open?
 - The pathname of the file is traversed, starting at root
 - Inode of root is known, to bootstrap the traversal
 - Recursively do: fetch inode of parent directory, read its data blocks, get inode number of child, fetch inode of child. Repeat until end of path
 - If new file, new inode and data blocks will have to be allocated using bitmap, and directory entry updated

Reading and writing a file

- For reading/writing file
 - Access in-memory inode via file descriptor
 - Find location of data block at current read/write offset
 - Fetch block from disk and perform operation
 - Writes may need to allocate new blocks from disk using bitmap of free blocks
 - Update time of access and other metadata in inode

Disk buffer cache (1)

- Results of recently fetched disk blocks are cached
 - LRU to evict if cache is full
- File system issues block read/write requests to block numbers via buffer cache
 - If block in cache, served from cache, no disk I/O
 - If cache miss, block fetched to cache and returned to file system
- Writes are applied to cache block first
 - Synchronous/write-through cache writes to disk immediately
 - Asynchronous/write-back cache stores dirty block in memory and writes back after a delay

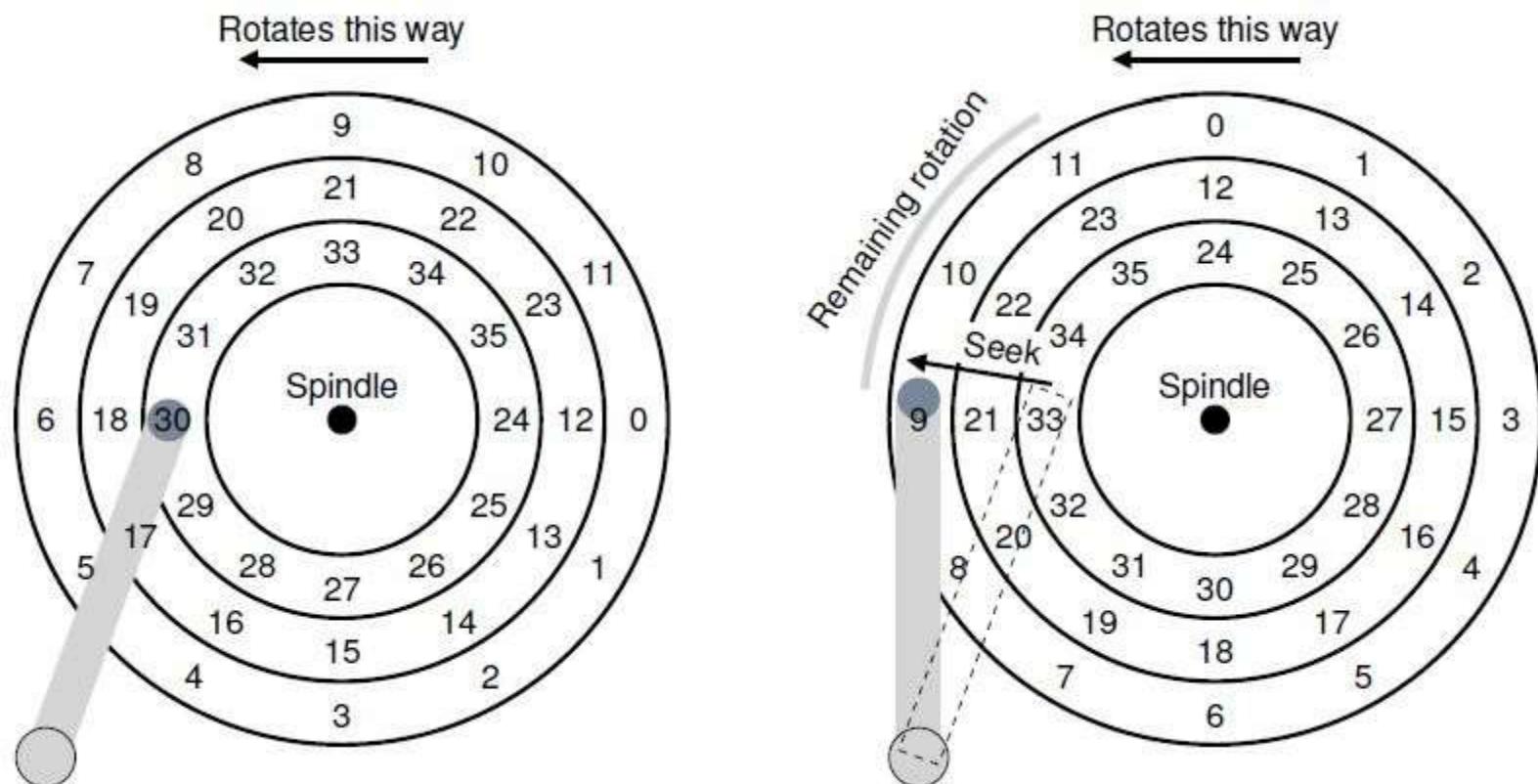
Hard disk internals

Hard disk

- Interface: a set of 512-byte blocks (sectors), that can be read or written atomically
- Internals: one or more platters, connected by a spindle, spinning at ~10K RPM (rotations per minute)
- Each platter has a disk head and arm
- A platter is divided into multiple tracks, and each track into 512-byte sectors

What happens when accessing a sector?

- Suppose disk head at 30, need to access 11
- Seek to the correct track, wait for disk to rotate



Time taken for I/O operation

- Time taken to read/write a block consists of
 - Seek time to get to the right track (few ms)
 - Rotational latency for disk to spin to correct sector on the track (few ms)
 - Transfer time to read sector (few tens microsec)
- Given high seek and rotational latency, usually rate of sequential access > rate of random access

	Cheetah	Barracuda
$R_{I/O}$ Random	0.66 MB/s	0.31 MB/s
$R_{I/O}$ Sequential	125 MB/s	105 MB/s

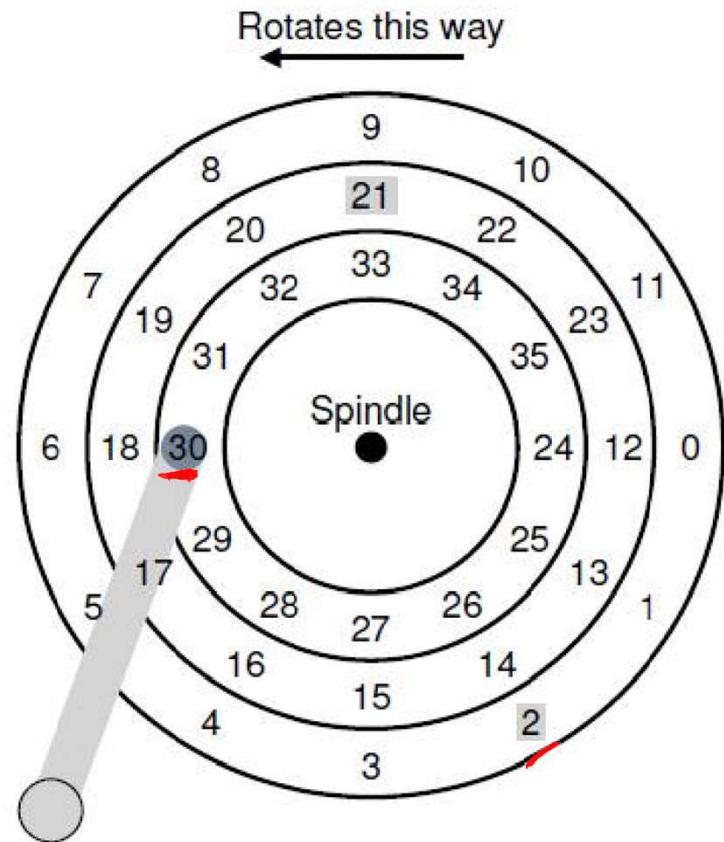
Figure 37.6: Disk Drive Performance: SCSI Versus SATA

Disk Scheduling

- Requests to disk are not served in FIFO, they are reordered with other pending requests
- Why? In order to read blocks in sequence as far as possible, to minimize seek time and rotational delay
- Who does scheduling? OS does not know internal geometry of disk, so scheduling done mostly by disk controller

Shortest Seek Time First (SSTF)

- Access block that we can seek to fastest
 - Go to 21 (one track away) before 2 (two tracks away)
- Problem: starvation
(some requests that are far from current position of head may never get served)

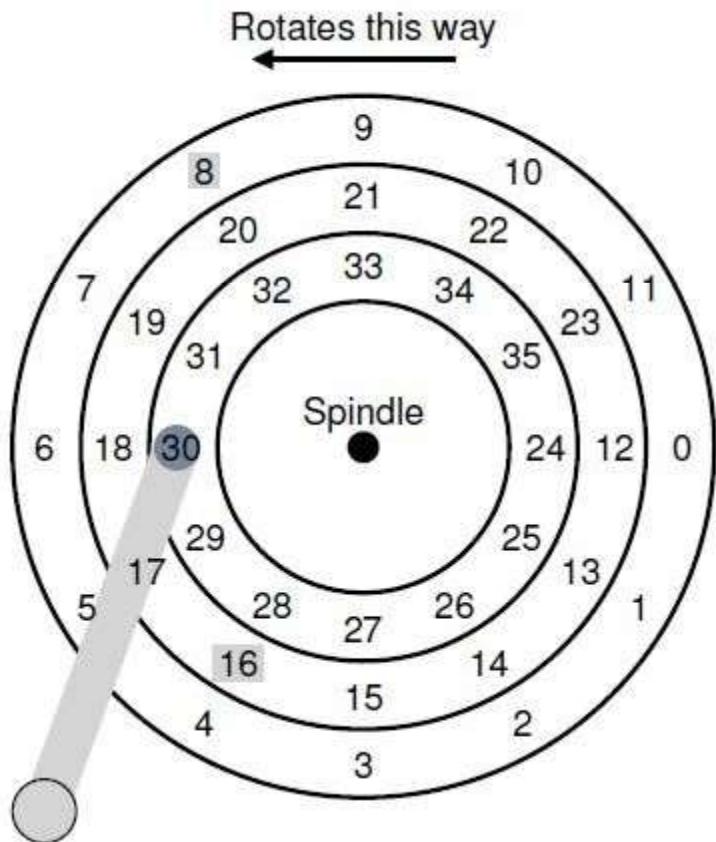


Elevator/SCAN algorithm

- Disk head does one sweep over tracks and serves requests that fall on the path
- Elevator/SCAN: sweep outer to inner, then inner to outer
- C-SCAN: sweep only one direction (say, outer to inner) and circle back, start again
 - Why? Sweeping back and forth favors middle tracks more
- F-SCAN: freeze queue while scanning
 - Why? Avoid starving far away requests

Shortest Positioning Time First (SPTF)

- Considers both seek time and rotational latency
 - Better to serve 8 before 16, even though seek time is higher
 - Why? 16 incurs a much higher rotational latency



Data storage on disk

- Bits stored on disk with some error detection/correction bits
 - Correct random bit flips
 - Detect corruption of data
- Disk controller or OS can handle some errors (e.g., blacklisting certain sectors)
- If errors cannot be masked, user perceives hard disk failures
- Technologies such as RAID (Redundant Array of Inexpensive Disks) provide high reliability and performance by replicating across multiple disks.

5 my a disk has 200 cylinders from 0 to 199
disk head is currently at 53

Pending request

98 193 37 122 14 124 65 67

< acknowledge the total mismatch , now head
will reverse

FCFS

D 11 37 53 65 67 74 122 124 183 199

53

94

123

37

14

122

67

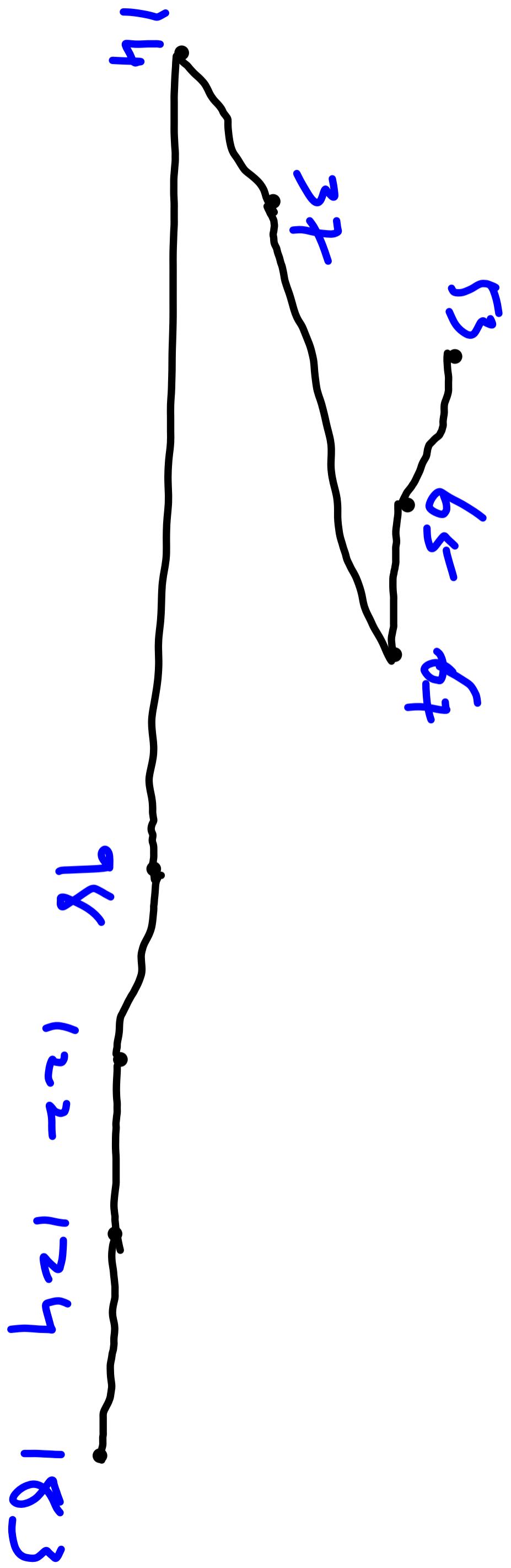
124

(182-53)+(183-14) 165
(183-37)+(121-37)

+ (122-14)+(124-14)+(124-65)+
(67-15)= 410

SSTF

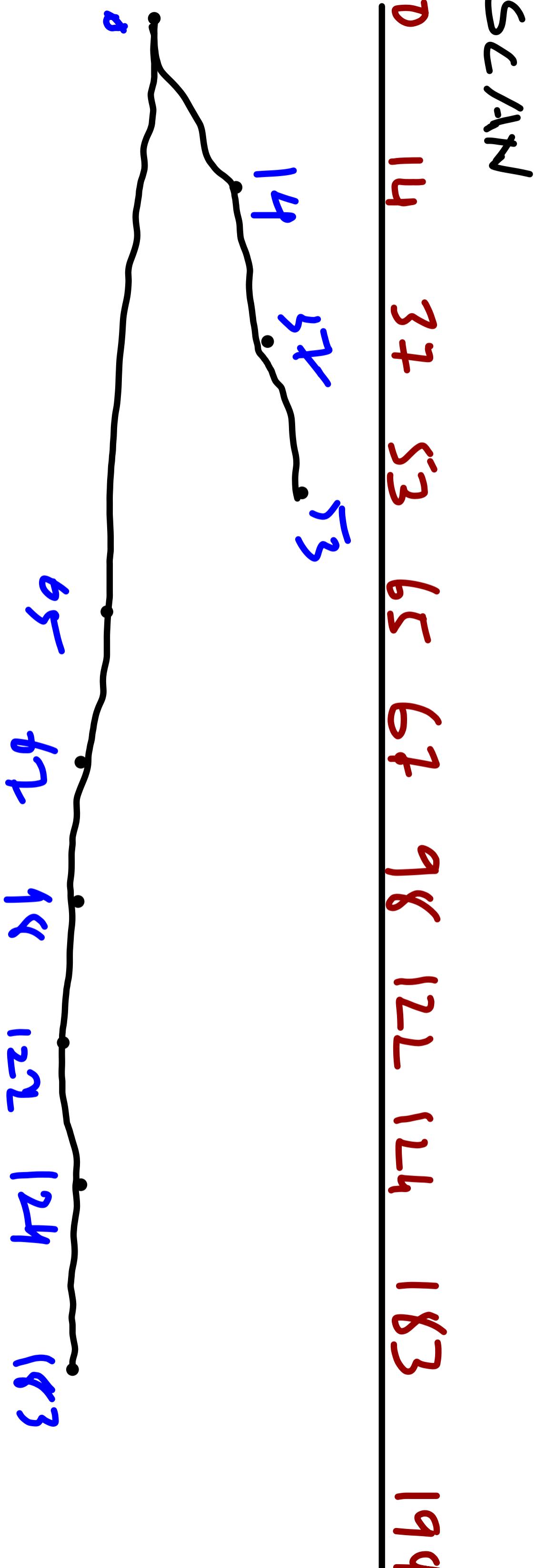
0 14 37 53 65 67 18 122 124 183 199



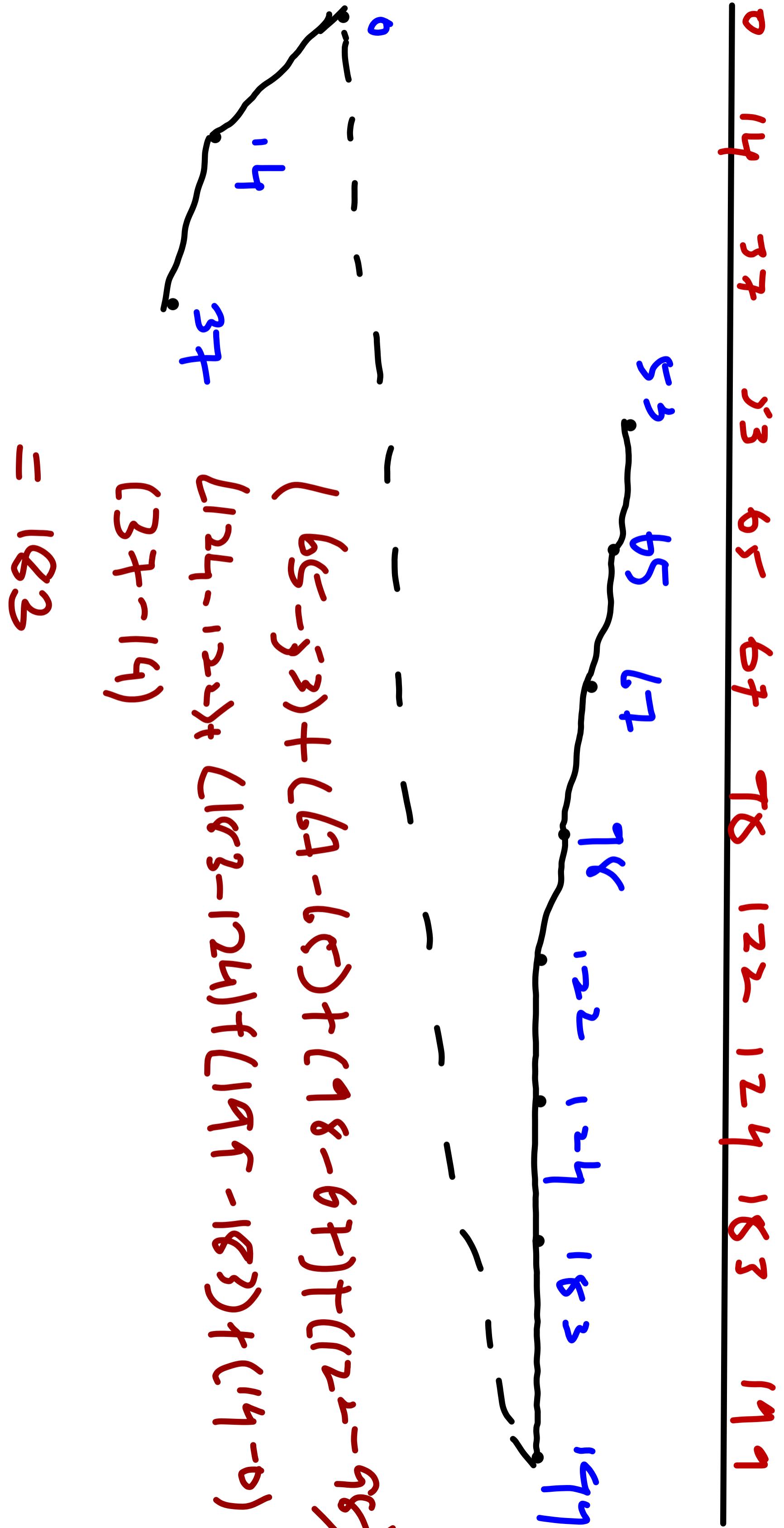
$$\begin{aligned} & ((15 - 53) + ((17 - 65) + (67 - 37) + (37 - 14)) \\ & + (122 - 124)) + ((124 - 122) + (183 - 125) \\ & = 236 \end{aligned}$$

= 236

$$(53-37) + (37-14) + (14-0) + (65-6) + (67-65) + (98-67)$$
$$+ (12-9) + (24-12) + (83-24)$$



C-SCAN



C - $\log k$

0 14 37 53 65 82 122 124 163 191

ζ_3 b_3 b_7 b_{11} b_{13} b_{17} b_{19} b_{23}

b_{14}
 $(65 \cdot \zeta_3) + (b_{14} \cdot \zeta_3) + (18 - b_7)$
 $+ (121 - 98) + (124 - 122) + (153 - 124)$
 $+ (37 - 14)$

= 153

File allocation method:

- Space to file
- disk is utilized

efficient manner.

Factors:

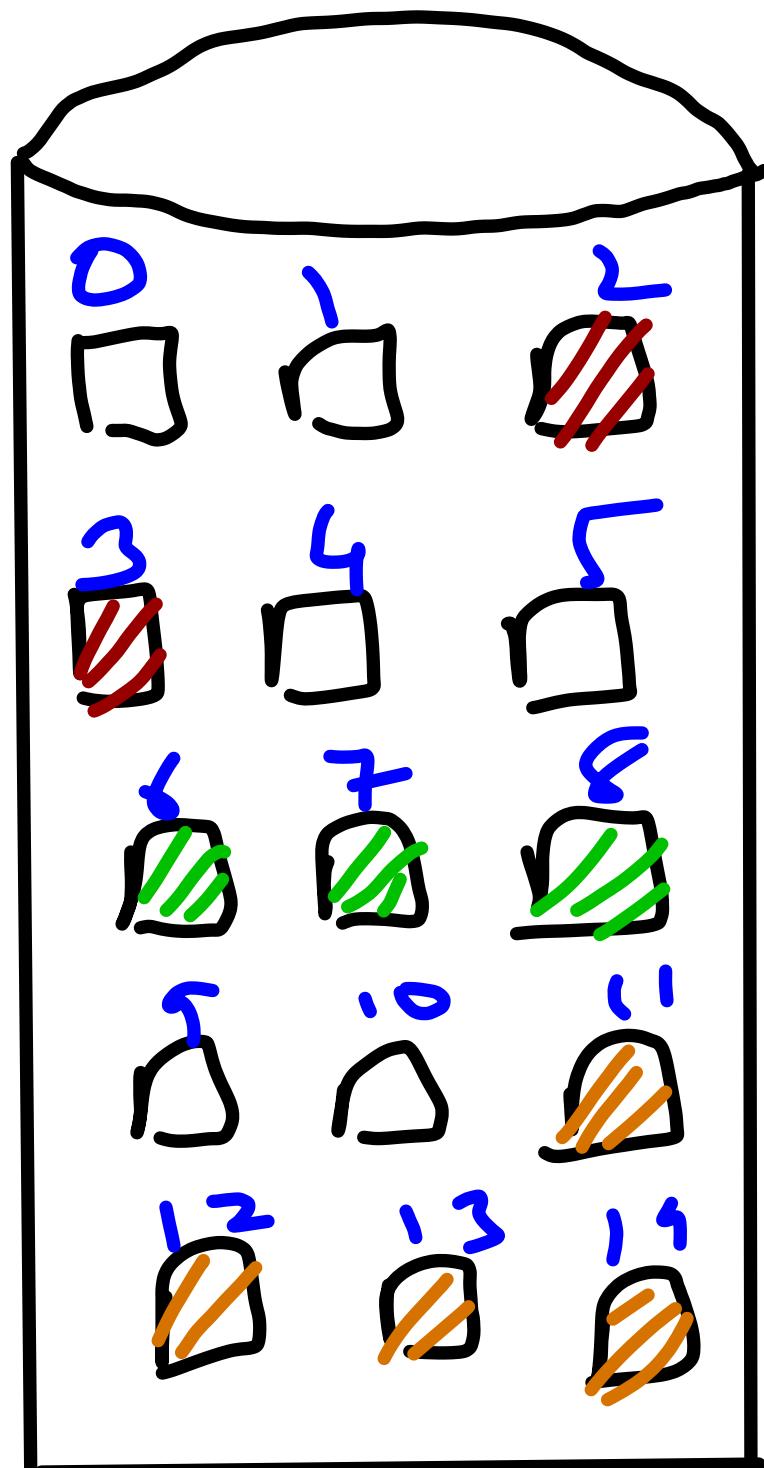
- Processing speed
- disk space utilization

different types of allocation methods.

Contiguous Allocation:

File - set of contiguous addresses on disk

- Linear ordering



Directory

file	start	len
A	2	2
B	6	3
C	11	4

- Sequential/random access are supported.

Random access

C → has 4 records
to access 1st record

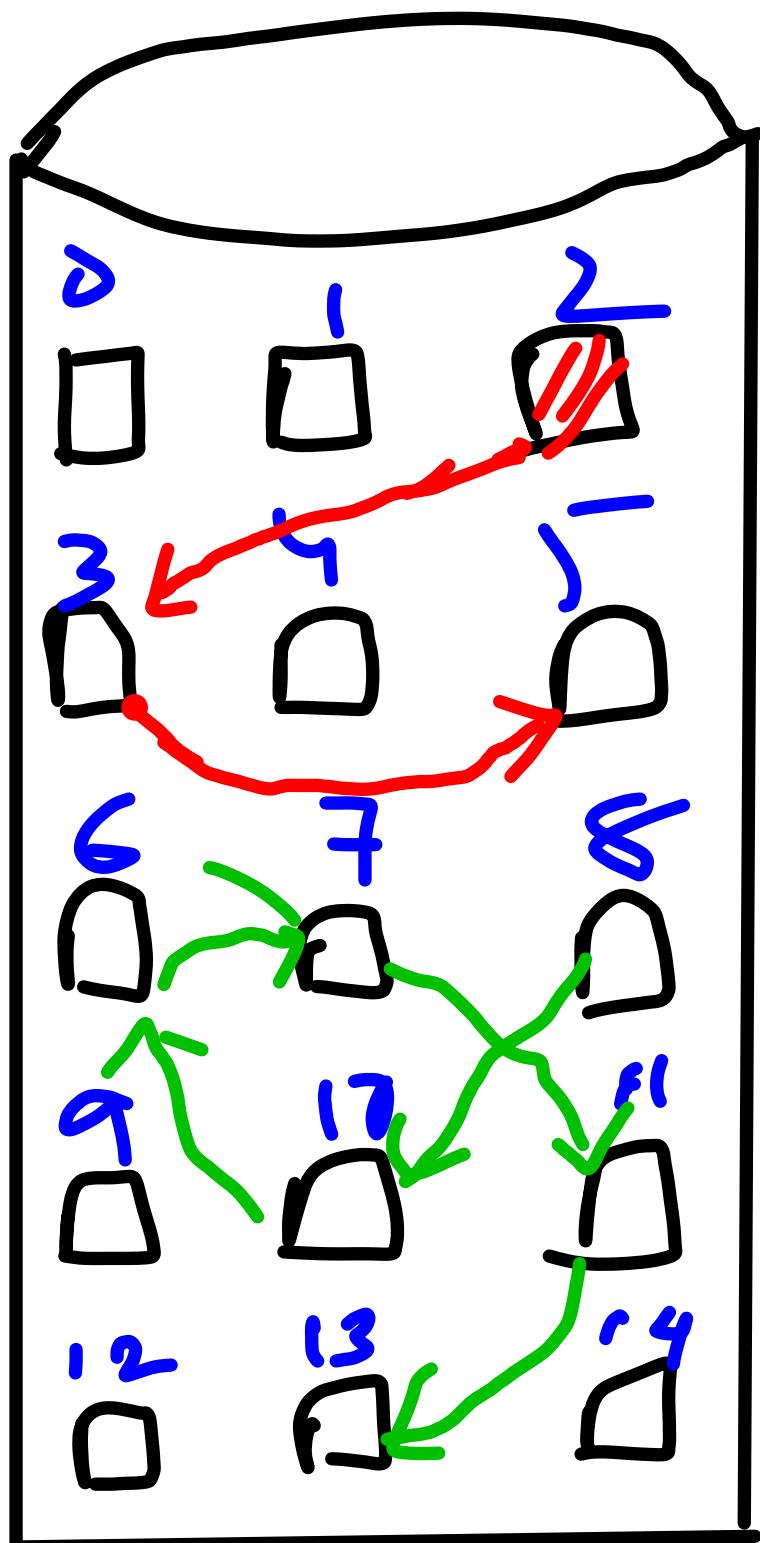
$$11 + 1 = 12$$

Disadvantages:

- Space for New file
- External fragmentation

Linked list:

- linked list of blocks
- No external fragmentation
- used only for sequential access of file.



tile	start	end
A	2	5
B	8	13

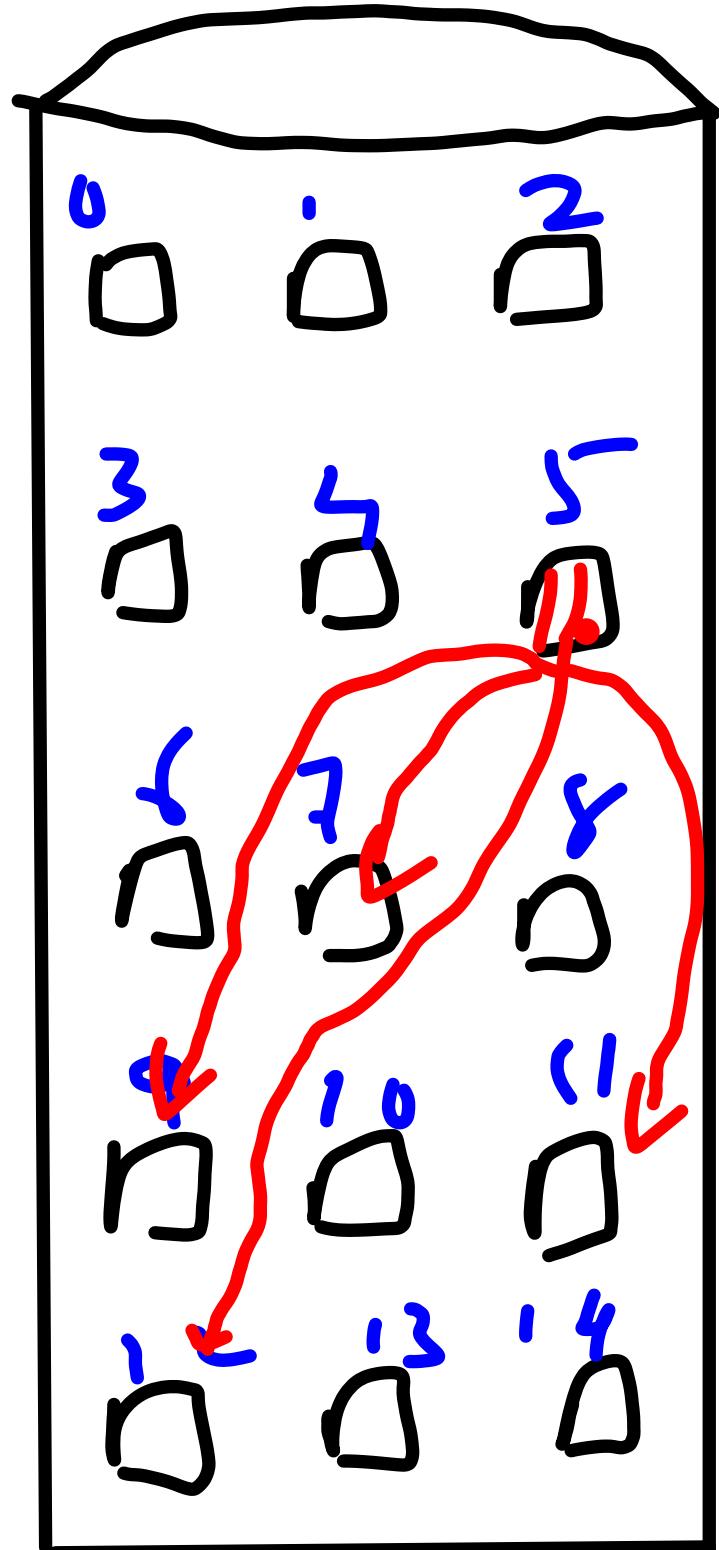
Indexed

- solves problem of linked list
- allows random access

Index block

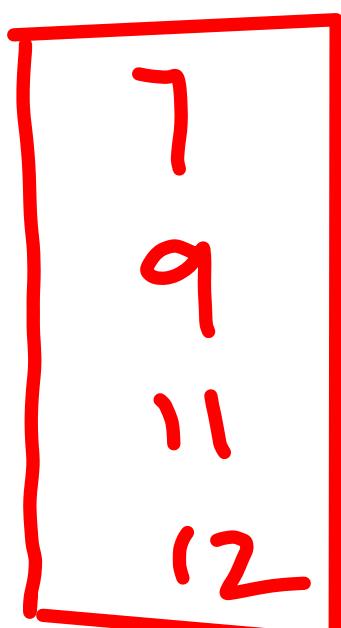
all pointers at one location

Each file has its own index block.



file	index block
A	5
C	10

A → 7 9 11 12
Pointers to the blocks
are stored in index block.



.