CS 111 Notes

Rajiv Anisetti

Fall 2018

1 Chapter 1: A Dialogue on the Book

The construction of operating systems is dependent on three key ideas:

- virtualization
- concurrency
- persistence

2 Chapter 2: Introduction to Operating Systems

When a program runs, the processor is **fetching**, **decoding**, and **executing** instructions from memory millions of times per second. This is known as the **Von Neumann** model of computing. The primary goal of the OS is to make the system **easy to use**. The main way in which an OS does this is through **virtualization**.

> **Virtualization:** taking a *physical* resource (such as the processor, memory, or disk) and transforming it into a *virtual*, more powerful version of itself

Because of the OS' use of virtualization, it is commonly called a **virtual machine**. In order for the OS to allow its users to manipulate the virtual resources, it provides **system calls**. An OS' compilation of system calls to manipulate the virtual machine is called its **standard library**. As a result, the OS essentially becomes a **resource manager**, where the resources are the system mechanisms such as the CPU, memory, and disk.

2.1 Virtualizing the CPU

Virtualizing the **CPU** makes it possible have one processor be able to perform a multitude of unrelated tasks, seemingly duplicating the processor. Because of this, it is possible to run multiple programs concurrently through what seems to be different processors, even on a single-processor machine.

Essentially, **virtualizing the CPU**, creates the illusion that a system has many processors. But with this power comes the issue of which instructions to execute/process in what order. This is decided by **policies** within the OS, which contribute to the overall **mechanisms** of the OS.

2.2 Virtualizing Memory

Physical memory is modeled as an array of bytes. It is possible to **read** and **write** to physical memory by specifying the address

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

Figure 1: An example program

Take a look at the above program. It simply increments a variable every second referenced by a pointer once every second. At the beginning of the program, it prints out the address that the pointer points to. Now let's take a look at the output when running two instances of this program concurrently, shown below.

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

Figure 2: Output of example program

The output shows each instance of the program incrementing their counts concurrently but independently, even though the address location for both variables was the same. This is possible due to **virtualizing memory**. Each of the two processes has its own **virtual memory** that the OS maps to the system's **physical memory**. Even though both processes address the same location, they don't run into each other due to the fact that their **virtual address spaces** are disjoint, and each process essentially has its own private memory.

2.3 Concurrency

Concurrency is another big topic within the field of operating systems. When an OS is juggling so many processes *concurrently*, it is easy to get mixed up and run into problems. Such problems can be illustrated through the same issues encountered when attempting to **multi-thread** a program. Let's take a look at the issues we can come across when attempting to multi-thread in the below program.

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

Figure 3: A multi-threaded program

The program simply creates two threads and assigns the *worker* function to them, which simply increments the *counter* variable the number of times that the *loops* variable is set to. When running this program with *loops* set to 1000, we would expect the final value to be 2000 (in fact, the final value should always be two times the value of the input, as there are two threads incrementing the counter. The output of such a call returns a final value of 2000 as expected, shown below.

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

Figure 4: A multi-threaded program

Now let's take a look at the output when we set *loops* to be a higher value, such as 100000, shown below. We would expect the final value to be 200000, but this is not the case. Running the program twice, we get not only incorrect final values, but *different* final values!

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

Figure 5: The same program run with *loops* set to 100000

This discrepancy is due to the OS workflow for incrementing the counter variable. It takes three **instructions** for the OS to do so. First, it must **load** the variable from memory, **increment** it, and then **store** it back into memory. When the threads are doing so, these three steps don't happen **atomically** (or all at once), so undefined behavior can result. Thus is the problem of **concurrency**.

2.4 Persistence

Data stored inside system memory is quickly accessed, but also very **volatile**. System devices such as DRAM and caches are susceptible to memory/data loss when power is shut off or the system crashes. Because of this, we need a way to store data **persistently**, and for this we need hardware as well as software.

The hardware usually comes in the form of a **hard drive** or solid-state drive **(SSD)**. These devices are capable of holding onto data for long term periods. The way such devices are accessed is through the **file system**, which is responsible for storing any **files** created by the user in an efficient and sustainable manner. **System calls** such as *open()*, *read()*, and *write()* are routed to the file system.

Unlike system memory, the OS does *not* virtualize the disks. This is due to the assumption that the user would want to **share** information within the files.

> **Example:** when a user writes a C program and compiles it, it is possible through **sharing** files. First, the user writes the source code, then uses a compiler to create an executable that can be run by the user at any point. This is done by sharing the data in the files throughout the disk.

There are many different ways to persist data. Many file systems delay writes in order to reduce overhead and try to write in batches. In addition, these systems also have write protocols, such as **journaling** and **copy-on-write** to ensure successful recovery to a reasonable state after a failed write. All of these mechanisms are efforts spent towards implementing **persistence**.

2.5 Design Goals

The basic goals of **virtualization**, **concurrency**, and **persistence** are integral to the design of all operating systems, but there are other goals we must take into account when designing an operating system.

The first of these would be **abstraction**. Through abstraction, we are able to build large scale programs and not have to worry about assembly language or transistors. Abstraction is fundamental to everything done in computer science! Thus, creating some level of abstraction is crucial as it **allows us to build larger scale architectures without having to worry about low-level components**.

The next goal is **performance**. In other words, we want to build an OS that **minimizes overhead**. Virtualization and facilitating system use are wroth it, but not at any cost. As a result, **it is crucial that the overhead created by such endeavors tolerable and as optimal as possible**.

Another goal is **protection** between applications. It is important that malicious or harmful behavior from one application does not cause another to fail. Thus, there must be an appropriate level of **isolation** between applications.

The last design goal to discuss is **reliability**. The OS must always be running, as if the OS crashes then *every application will crash*. Everything running on the system is highly dependent on the OS. Thus, **operating systems must be designed with a high level of reliability**.

Other goals such as energy-efficiency, mobility, and security are also critical depending on the circumstances. Each OS is differently designed to address such goals and function on a different set of devices.

3 Chapter 3: A Dialog on Virtualization

Omitted due to breadth and irrelevance.

4 Chapter 4: The Abstraction: The Process

Even though there are limited central processing units (CPUs), the OS must create the illusion that there are many CPUs by **virtualizing** them. This is done through **time sharing** of the CPU, where the OS stops processes in order to run another on the same CPU. The tradeoff for this is **performance**. *Running many processes will make each each one run slower.*

In order to virtualize the CPU, the OS must implement low-level **mechanisms** and higher level **intelligence**. This is done through **mechanisms** and **policies**, respectively.

A **mechanism** is a low level protocol that implements a needed functionality. One example is a *context switch*, which is a mechanism that allows the OS to stop running one program and run another on a CPU.

A **policy** is a an algorithm for making a decision for the OS. For example, a *scheduling policy* would decide which program gets to run on a certain CPU, and makes this decision based on many factors such as call history, workload metrics, and performance metrics.

4.1 Processes

A **process** is the abstraction provided by the OS for running a program. A process contains a **machine state**, which is what a program can read or update while it is running. Several components of the machine state are *memory* (or the program's address space), *registers* (such as the instruction pointer/program counter stack pointer), and *I/O information* (such as which files the program has open).

4.2 Process API

Each OS must implement process APIs, which are available through the OS interface. In some form, all modern OS's implement the following APIs.

- **Create:** an API for creating a process.
- **Destroy:** an API for destroying a process by force (though many processes will exit by themselves peacefully).
- **Wait:** an API for waiting for a process to finish
- **Miscellaneous Control:** other control processes, such as suspending a process until it can continue later.
- **Status:** an API for getting status information about a process, such as its state or how long it has run.

4.3 Process Creation in More Detail

The procedure for creating a process can be summarized in a couple steps.

1. **Load:** First the OS must load the code of the program and any static data from disk to the address space of the process. Programs reside on the disk in **executable format**, so the OS must read the program code/data into memory from disk.
2. **Allocating Memory:** After loading the program, the OS must allocate necessary memory onto the **stack** (where local variables are stored in C) and the **heap** (where dynamically allocated variables are stored).
3. **Other Initialization Tasks:** The OS will perform other initialization tasks before running, such as opening the three file descriptors for each program (stdin, stdout, stderr).

After all of this initialization, the OS is ready to delegate the running of the process and all further action to the CPU setting the program to run at the *main()* function.

4.4 Process States

A process can fall into several states, discussed below.

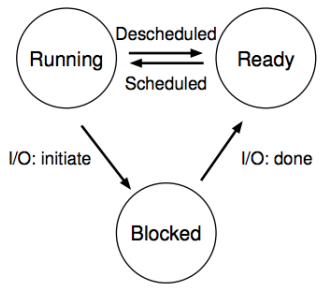


Figure 6: Process State Diagram

- **Running:** the process is currently running on a CPU and executing instructions.
- **Ready:** the process is ready to run, but has not been chosen to run yet by the OS for whatever reason.
- **Blocked:** the process has performed an operation that has caused it to pause its execution until a further action validates to process being ready again, such as an I/O request.

Below is table of how an OS might schedule two processes.

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 7: Tracing Process State w/ I/O

4.5 Data Structures

The OS holds several data structures for tracking processes, one of which is a **process list**. Below is an example of the information that the *xv6 kernel* holds in order to keep track of processes.

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

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

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

Figure 8: Process Data Structure for the xv6 Kernel

Notice that the process keeps track of the **register context** for each process through the context struct. When the OS halts a process and allows the CPU to begin running another process, it saves the **context** of the original process through this struct and places these values back into the physical registers when the process is reinstated. This is part of a procedure called a **context switch**.

In addition to the register context, the OS typically keeps track of **states** as well. For example, when the process is being initialized, it is typically in the **initial** state. When a process is finished running, but hasn't been cleaned up, it is in its **final state** (in UNIX, this is known as **zombie** state). This state is useful for when a **parent** process wants to see how a **child** process has returned, perhaps using the *wait()* function.

Another way to address the data structure used for handling processes is the **Process Control Block (PCB)**.

5 Chapter 5: Process API

5.1 The *fork()* System Call

```
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) {         // fork failed; exit
9          fprintf(stderr, "fork failed\n");
10         exit(1);
11     } else if (rc == 0) { // child (new process)
12         printf("hello, I am child (pid:%d)\n", (int) getpid());
13     } else {              // parent goes down this path (main)
14         printf("hello, I am parent of %d (pid:%d)\n",
15                 rc, (int) getpid());
16     }
17     return 0;
18 }
19
```

Figure 9: Example call to *fork()*

The ***fork()*** system call is used to create a new process. The **child** process is almost an exact copy of the **parent** process, with its own copy of the address space, its own PC, registers, etc. However, the child process now acts independently of the parent process and executes code *not from main(), but immediately after the fork() call*, as if it made the call itself.

fork() returns 0 to the child process, and returns the **process identifier (PID)** of the child process to the parent process. This is useful for tracking the completion/status of the child process.

It is important to note that the output of a multi-process program is **not deterministic**. That is, there is no defined order as to whether the parent or child process executes instructions in which order. This can lead to problems similar to those of **multi-threading**.

5.2 The *wait()* System Call

The *wait()* system call (or *waitpid()* is useful for delaying execution until a child process finishes. This is helpful in making program output **deterministic**. If the *wait()* function is used, the parent process waits until the child process has finished in order to continue executing.

5.3 The *exec()* System Call

The ***exec()*** system call is useful when you want to run a program that is different from the calling program. *exec()* works in the following way.

Given the name of an executable and its arguments, it **loads** code and static data, overwriting the current loaded code and static data of the calling program. In addition, it reinitializes the stack and heap to adhere to the new program. After this, it runs the named program with the arguments specified as the *argv* of the new program, basically transforming the calling process (or child process of it) into the new process. Take a look below for a sample program that calls *fork()*, *wait()*, and *exec()*.

```
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 }
28
```

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

Figure 10: Program calling *fork()*, *wait()*, and *exec()*

5.4 Why? Motivating the API

The disjoint aspect of *fork()* and *exec()* is incredibly important in building a shell, due to the freedom to execute code between forking and executing. This allows the implementation of several different options before executing such as **redirecting output from stdout to a file**. As a result, *fork()* and *exec()* can be immensely powerful tools.

A **shell** is simply a user program. It shows a **prompt** and then waits for the user to input a command (typically an executable followed by arguments/options). The program then locates the executable and *fork()*'s and then has the **child** process run the executable with the arguments. After this, it *wait()*'s until the command is complete, then prompts the user to enter another command.

A **pipe** is a way of connecting the output of one process to the input of another, in turn creating long chains of command. This is immensely useful for shells, and is made possible through the *fork()* and *exec()* calls along with their disjoint nature.

5.5 Process Control and Users

There are more ways of interacting with processes than through the system calls mentioned above. In particular, users can send **signals** to processes to trigger specific control flows. For example, the *kill()* system call can be used to issue a signal to a process, telling it to pause, die, or other imperatives.

The signal subsystem provides an easy way of delivering external events to processes. In order to use this system, a process should use the ***signal()*** system call to catch signals. When a process, catches a signal that has been warned through the *signal()* call, it will suspend further execution to execute the code specified inside the handler function passed to *signal()*.

Because of the immense power of signals, it is important to keep a sharp notion of **user** control. A user should not be able to arbitrarily send signals to any process within a system, as many users could be using the system. Because of this, many systems incorporate a login functionality where users can then launch processes and exercise full control over them (but cannot signal other users' processes). It is then the job of the OS to distribute system resources to the user.

6 Chapter 6: Mechanism: Limited Direct Execution

The OS must virtualize the CPU in an efficient manner while maintaining control over the system. This must be done through efficient **time sharing** of the CPU that doesn't create too much overhead.

6.1 Basic Technique: Limited Direct Execution

OS developers came up with the idea of **limited direct execution** with speed in mind. The "limited" portion will be discussed later, but the "direct execution" portion is intuitive. This procedure consists of simply creating a process entry for the process list, loading the program and its static data, then jumping to the *main()* function and executing code until the **return** from *main()*. After this, control is given back to the OS. Below is an example time chart of direct execution.

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 11: Direct Execution (without Limits)

This procedure has several flaws.

- How do we make sure the process isn't doing anything we don't want it to do, while still running efficiently?
- How does the OS implement **time sharing**, where it can stop a process and begin executing another?

In answering these questions, we will develop the "limited" portion of **limited direct execution**.

6.2 Problem 1: Restricted Operations

Direct execution is fast, as the process has complete control over the CPU, but there is no way for the process to perform restricted operations such as I/O operations or gaining access to other system resources. As a result, **the OS must allow processes to perform restricted operations, without giving complete control over the system**.

If a process had complete control, many bad things could happen without intervention, such as writing to invalid memory locations or reading from protected locations. Because of this, OS's implement a new processor mode called **user mode**. When a process is running in user mode, it has limited permissions and is highly restricted in what it can do.

On the contrary, processes running in **kernel mode** can do what they like, including issuing I/O requests and executing restricted instructions.

Thus, in order for users to perform **restricted operations**, they must make **system calls**. System calls allow the OS to carefully expose restricted functionalities such as accessing the file system, allocating memory, etc.

In order to execute a system call, the system executes a **trap** instruction. This instruction jumps into the kernel and raises permissions to **kernel mode**, where the system performs the privileged operations, and then issues a **return-from-trap** instruction where the system jumps back into the calling program and reduces privileges to **user mode**. In order to ensure correct execution when returning to the process, the OS pushes the program counter, registers, and other data of the program onto a **kernel stack**, where it can pop these values off and continue execution after a **return-from-trap** instruction.

But how does the OS know what code to execute once inside kernel mode? If the user were able to simply issue an address, then this would be *very dangerous*, as any code could be executed with kernel privileges. Because of this, the kernel sets up a **trap table** at boot time. The kernel boots up in **kernel mode** and configures the the hardware to tell it what could should run when an exceptional event occurs. The OS informs the hardware of these **trap handlers** and it remembers the locations of these handlers until the next reboot. Below is an example control flow for **Limited Direct Execution**.

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

Figure 6.2: **Limited Direct Execution Protocol**

Figure 12: Limited Direct Execution Control Flow

6.3 Problem 2: Switching Between Process

The next question is **how can the operating system regain control of the CPU in order to switch between processes?**

6.3.1 A Cooperative Approach: Wait for System Calls

One approach, known as the **cooperative approach**, is to assume that processes within the systems behave reasonable and trusts them. In this utopian world, processes give up control through **system calls**, where the OS can issue a **trap** instruction. In addition, when invalid/illegal operations are attempted, the OS can also **trap** and regain control (and most likely kill the process).

6.3.2 A Non-Cooperative Approach: The OS Takes Control

The cooperative approach has critical downfalls. For example, what if a program gets stuck in an infinite loop that doesn't make any system calls? The OS would never regain control, and the only solution here would be to **reboot the machine**.

Thus, many OS's implement a **timer interrupt**, which is basically a timer that interrupts execution of a process every cycle in order to give control back to the OS. When the interrupt is sent, the OS executes its preconfigured **interrupt handler**. The OS configures this timer interrupt during boot time, and *once it starts this timer, it can be sure it will eventually regain control even in the case of non-cooperative programs.*

6.3.3 Saving and Restoring Context

When the OS regains control of the CPU, it needs to know whether to continue execution of the halted process or to execute another process. This decision is made by the **scheduler**.

If the decision is made to switch, then the OS must perform a **context switch**. This process is relatively simple. The OS must save some register values for the currently running process (onto its kernel stack) and then store values from the to-be-executed process (found on its kernel stack) to the registers. The OS then changes the stack pointer to point to the new process' kernel stack, so that when the **return-from-trap** instruction is executed, it is almost as if the new process issued the **trap**. Below is a diagram illustrating the *switch()* process.

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

Figure 13: Limited Direct Execution Protocol

6.4 Concurrency Issues

What happens when a **timer interrupt** occurs during the execution of a **system call**? Such **concurrency** issues are handled differently by each OS. One way to handle such issues is to **disable interrupts** during interrupt processing. Operating systems have also developed several **locking** systems to protect concurrent access to internal data structures. These will be touched upon when we discuss more about concurrency.

7 Chapter 7: Scheduling: Introduction

Now that we have discussed low-level **mechanisms**, it is important to discuss high-level **policies** as well. We will now discuss a series of **scheduling policies** (also known as **disciplines**) that have been developed over the years.

7.1 Workload Assumptions

Let's make some assumptions about the processes running on our system (otherwise known as the **workload**).

- Each job takes the same amount of time
- All jobs arrive at the same time
- Once started, a job runs to completion
- All jobs only use CPU (no I/O)
- The run-time of each job is known

Though these are unreasonable assumptions, we will relax them as the section goes on and will eventually be able to build a **fully-operational scheduling discipline**.

7.2 Scheduling Metrics

The main **metric** we will use for analysis will be **turnaround time**. Turnaround time is defined as the time of completion of the job minus the time that the job arrived.

$$T_{turnaround} = T_{completion} - T_{arrival}$$

This is a **performance metric**. Another type of metric is **fairness**, which is a measure of how many jobs were able to run. Performance and fairness are often at odds with one another.

7.3 First In, First Out (FIFO)

The first scheduling discipline, and most basic, is **first in, first out**. This algorithm is very simple, and works well given out assumptions. Let's take a look at a simple case.

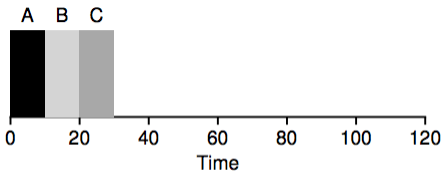


Figure 7.1: **FIFO Simple Example**

For this case, our algorithm works well. Our average turnaround time is $\frac{10+20+30}{3} = 20$. But **what happens when we take away our first assumption that all jobs take the same amount of time**. Take a look at the following example.

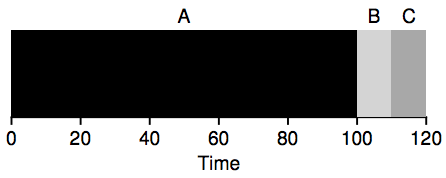


Figure 7.2: **Why FIFO Is Not That Great**

In this example, our **average turnaround time** increases dramatically. It is now $\frac{100+110+120}{3} = 110$! One large job causes the other shorter jobs to wait in line for a long period of time, known as the **convoy effect**. So how do we fix this?

7.4 Shortest Job First (SJF)

One way of fixing this would be to execute the **shortest jobs first**. A simple example is shown below.

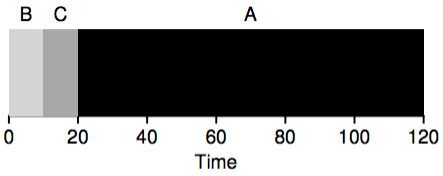
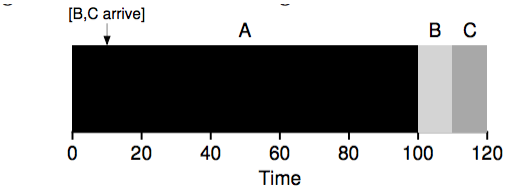


Figure 7.3: **SJF Simple Example**

Our turnaround time is now $\frac{10+20+120}{3} = 50$. This is a great improvement from FIFO, but let's **get rid of our second assumption that all jobs arrive at the same time**. Take a look at the more complicated following example.



In this example, jobs B and C arrive a little after A, but A is scheduled and goes to completion, taking our average turnaround back to $\frac{100+(110-10)+(120-10)}{3} = 103$. How do we fix this?

7.5 Shortest Time to Completion First (STCF)

In order to address the previous concern of having jobs arrive at different times, **we need to relax our assumption that jobs execute until completion**.

Now, let's assume we can **preempt** a job - that is, pause the job and save the rest of its execution for later. Now, with this **preemptive** scheduling, we can execute an algorithm that is **shortest time to completion first (STCF)**. Let's take a look at a simple case.

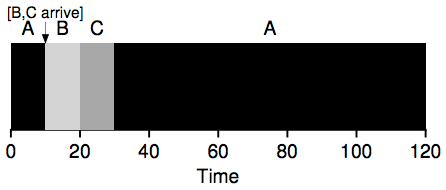


Figure 7.5: **STCF Simple Example**

In the above diagram, jobs B and C arrive at $t = 10$, but their times to completion are shorter than A, so we **preempt** A and finish jobs B and C before going back to finish A. Our new average turnaround time is $\frac{(20-10)+(30-10)+(120)}{3} = 50$! If turnaround time was our only metric, STCF would be a great choice, but more important metrics exist as well!

7.6 A New Metric: Response Time

With the introduction of time-shared machines, and the evolution of computing past batch computing, a new metric was born. Users were not interacting with the computers, demanding quick **response times**.

Response time is defined as the time at which the job was first scheduled minus its arrival time.

$$T_{response} = T_{scheduled} - T_{arrival}$$

The response time of the previous example would be $\frac{0+(10-10)+(20-10)}{3} = 3.33$, which is pretty bad. Imagine having to wait 10 seconds for a response from the terminal, because someone else's job got arbitrarily scheduled before yours. That's no good!

7.7 Round Robin

With **response time** in mind, the **Round Robin** scheduling algorithm was created. **RR** runs a job for a **time-slice**, then **preempts** the job and runs another job for a **time-slice**. Let's take a look at the following comparison between SJF and RR.

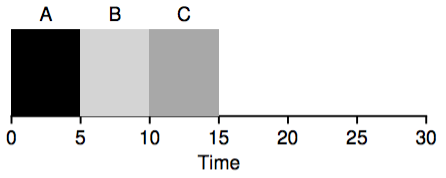


Figure 7.6: **SJF Again (Bad for Response Time)**

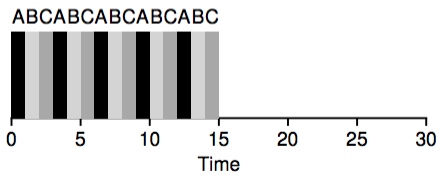


Figure 7.7: **Round Robin (Good For Response Time)**

The average **response time** for SJF in this example would be $\frac{0+5+10}{3} = 5$. The average response time for Round Robin would be $\frac{0+1+2}{3} = 1$. It is important to take into account the size of each time-slice, as

having too short of a time-slice would cause context-switching overhead to dominate our runtime. Thus, it is crucial to **amortize** the cost of context-switching by having larger time-slices and incurring less overhead.

Round Robin is far superior for response time, but there is a **trade-off**. Round Robin's average **turnaround time** takes a hit due to its prioritization of response time. This is true for many **fair** scheduling algorithms, that evenly distributes the CPU amongst all jobs.

7.8 Incorporating I/O

Now we must remove our fourth assumption, that processes do not make I/O requests. Let's take an example where process A makes several I/O requests, but process B makes no I/O requests. A foolish way of scheduling is shown below.

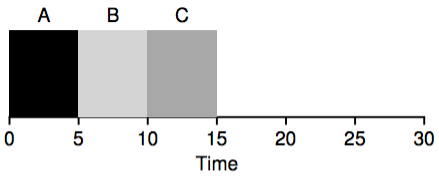


Figure 7.6: **SJF Again (Bad for Response Time)**

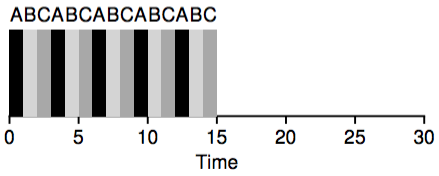


Figure 7.7: **Round Robin (Good For Response Time)**

A better way of utilizing our resources makes use of **overlap**. That is, **preempting** A while it is **blocked** due to its I/O request, and running a time-slice of the more CPU-intensive B until A is no longer blocked. Using this method, we are no longer wasting CPU time. Thus, **overlap is good for resource utilization**. A diagram is shown below.

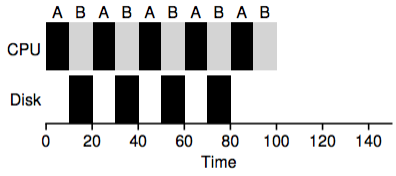


Figure 7.9: **Overlap Allows Better Use Of Resources**

7.9 Summary

So SJF is great for turnaround time, but RR is better for response time. This tradeoff between turnaround time and response time is common amongst operating systems. In the next section, we will relax our last assumption that the OS knows the length of each job, creating a **multi-level feedback queue**.

8 Scheduling: The Multi-Level Feedback Queue

The problem that the **multi-level feedback queue** attempts to solve is twofold: **how do we design a scheduler that minimizes response time *and* turnaround time *without having prior knowledge of job runtimes*?** Let's take a look at its implementation.

8.1 MLFQ: Basic Rules

The **MLFQ** operates through several distinct **queues**, each of which has a different priority level. Jobs that are on higher priority queues will run first, and if they are of the same priority, they will run **Round Robin**. Thus, the two basic rules of MLFQ are.

- **Rule 1:** if Priority(A) > Priority(B), A runs
- **Rule 2:** if Priority(A) == Priority(B), A & B run Round Robin

They key to MLFQ then resides in *how it sets priority*. In short, **MLFQ varies priority based on observed behavior**. If a job continuously issues I/O requests and relinquishes the CPU its priority will be kept high, but jobs that use the CPU for longer periods of time will move down in priority. Below is a static diagram of the queues.

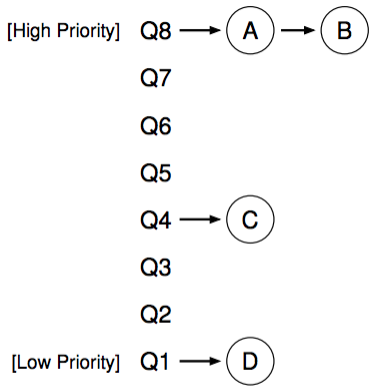


Figure 8.1: **MLFQ Example**

But with our current knowledge of MLFQ, wouldn't A and B just run forever Round Robin, with jobs C and D never getting any CPU time? Let's take a further look at how jobs **change priority** within MLFQ.

8.2 Attempt 1: How to Change Priority

Let's create some basic rules for changing priority.

- **Rule 3:** When a job enters the system, it is placed at the highest priority
- **Rule 4a:** If a job uses up an entire time-slice, it moves down in priority
- **Rule 4b:** If a job gives up the CPU before the time-slice, it *stays in the same priority level*

Let's take a look at how this scheduler would handle several cases.

8.2.1 A Single Long Job

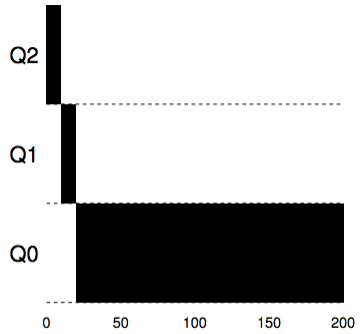


Figure 8.2: **Long-running Job Over Time**

Take a look at the above time diagram for a single job. Once it uses up an entire time-slice in each priority level, it moves down until it reaches the bottom-most priority where it runs to completion.

8.2.2 Along Came a Short Job

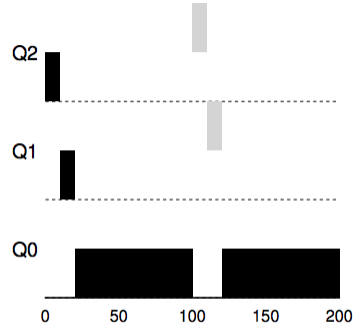


Figure 8.3: **Along Came An Interactive Job**

In this example, a shorter job comes in, where it only takes two time-slices to complete. It enters at top priority, finishes before reaching the bottom-most level, and then job A finishes.

Thus, *MLFQ doesn't know whether a job is short, but it assumes it is, and moves it to lower priorities if it isn't.*

8.2.3 What about I/O?

Let's take a look at when a job repeatedly issues I/O requests. As stated in our rules, it stays in our top-most priority level, as it relinquishes the CPU before a full time-slice. Thus, the next priority job gets to run (which in this case is our Q0 job).

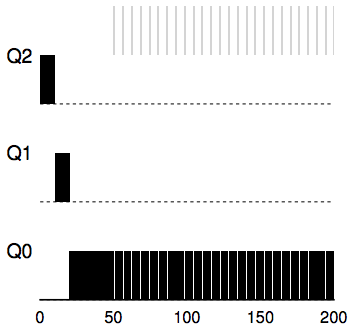


Figure 8.4: **A Mixed I/O-intensive and CPU-intensive Workload**

8.3 Problems With Our Current MLFQ

Our current model has three critical flaws.

- **Starvation:** if there are too many interactive jobs that issue I/O requests, they will run round robin forever, never allowing lower-priority jobs to run.
- **Gaming the System:** a user could easily game the scheduler by issuing a useless I/O request at the end of a time-slice, keeping unfair priority of the job
- **Changing Program Behavior:** a program can change behavior and become highly interactive, but in the current system it would be dominated by its past, staying in lower priority levels

8.4 Attempt 2: The Priority Boost

In order to solve problems 1 and 3, we can issue a **priority boost** to all jobs after a certain time S. With this implementation, jobs will no longer starve and can receive fair treatment if they change behavior and become interactive.

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

Let's take a look at how the time diagram changes. Notice that the two very interactive jobs no longer take up all CPU time running Round Robin.

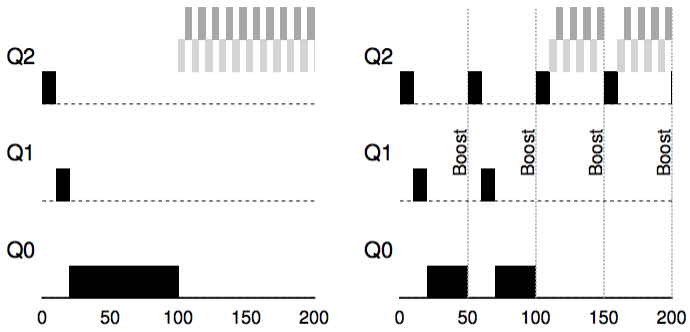


Figure 8.5: **Without (Left) and With (Right) Priority Boost**

Note: It is important to define S here properly. It is a **voodoo constant** - that is, its very confusing to set the value correctly. Too high, and programs will starve. Too low, and interactive programs won't get enough CPU time.

8.5 Attempt 3: Better Accounting

We still need to solve the problem of **gaming the CPU**. In order to do this, we can alter our fourth rule to have better **accounting**. Once a job has used up its time-slice at a certain priority level, it will be moved down in priority, *regardless of how many times it relinquished the CPU.*

- **Rule 4:** Once a job uses up its time allotment for a queue level (regardless of how many times it gave up the CPU), its priority is reduced.

Let's take a look at an example of someone trying to game the system with rule 4 now in place.

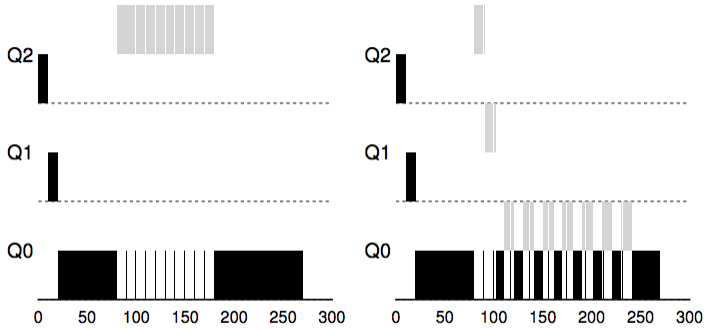


Figure 8.6: **Without (Left) and With (Right) Gaming Tolerance**

Now the interactive job is moved down in priority regardless of its I/O requests, making the the scheduler much more **fair**.

8.6 Tuning MLFQ and Other Issues

A leftover question is how should we **parameterize** the MLFQ scheduler. For example, how many queues should we use? How long should the time-slices be? Many MLFQs allow for varying time-slice length for different queues, with lower priority levels having longer time-slices. This is shown below.

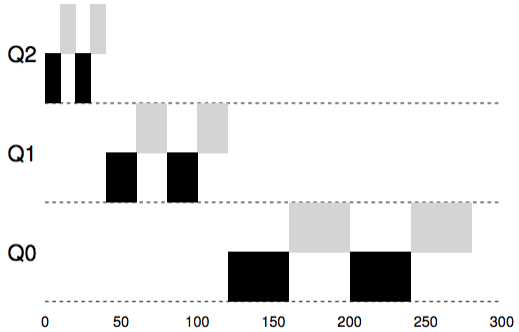


Figure 8.7: **Lower Priority, Longer Quanta**

Parameterizing the MLFQ is difficult, and takes knowledge of the **workload** and fine tuning to formulate answers to the questions above.

8.7 MLFQ: Summary

Here are our final rules for the **multi-level feedback queue**.

- **Rule 1:** if Priority(A) > Priority(B), A runs
- **Rule 2:** if Priority(A) == Priority(B), A & B run Round Robin
- **Rule 3:** When a job enters the system, it is placed at the highest priority
- **Rule 4:** Once a job uses up its time allotment for a queue level (regardless of how many times it gave up the CPU), its priority is reduced.
- **Rule 5:** After some time S, move all jobs to the topmost queue.

The MLFQ *uses history to predict the future*. It takes performance knowledge of a job to prioritize it accordingly, and because of this **it has the best of both worlds in terms of fairness and performance**.

9 Chapter 12: A Dialogue on Memory Virtualization

Omitted due to breadth and irrelevance.

10 Chapter 13: The Abstraction: Address Spaces

10.1 Early Systems

Before the **era of multi-programming**, OS's didn't create much abstraction, as there was no need. In terms of memory, the OS was simply a set of routines (or a library) that was placed in physical memory, and a running program used the rest of memory, as in the following example.

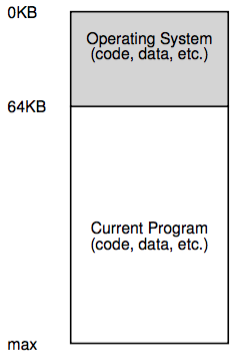


Figure 13.1: **Operating Systems: The Early Days**

10.2 Multi-programming and Time Sharing

As time progressed, people began **sharing** machines more, introducing the **era of multi-programming**. Soon after that, people started demanding even more of machines, and so the era of **time sharing** was born. But how did OS's manage memory when time-sharing? An early and foolish approach saved all of a process' state to disk and then loaded another process' state, but this is extremely slow due to the time-consuming disk reads/writes. A better approach is to keep the process in memory, but allocate a specific portion of memory to each process, as shown in the diagram below.

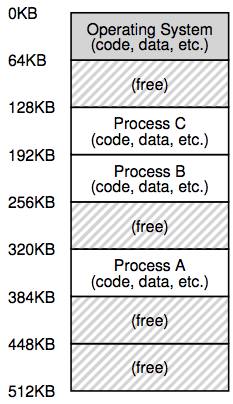


Figure 13.2: **Three Processes: Sharing Memory**

10.3 The Address Space

In order to efficiently manage memory while time-sharing, the OS must create an easy to use **abstraction** of physical memory. This abstraction is called a program's **address space**, or the program's limited view of the system's memory.

A program's address space contains all of the memory state of the running program. In particular, the address space contains the **code** of the program, the **stack** (where the program keeps all local variables as well as location in the function call chain, etc.), and the **heap** (where the program keeps all dynamically allocated memory). There is more static data as well, but we are only considering these three components for now. Take a look at how a sample address space might be organized.

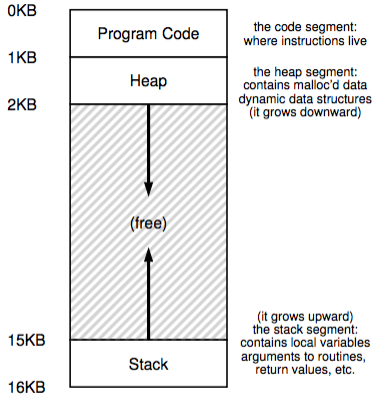


Figure 13.3: **An Example Address Space**

Notice how the **heap** and **stack** grow. The stack grows upward, while the heap grows downward within the address space. This is simply convention, and can be changed (as with **multi-threaded programs**, where this is no longer possible).

Thus, this is how the OS **virtualizes memory**. The address space is an **abstraction** of physical memory, and consists of **virtual addresses** that are then *mapped* to real physical memory locations.

10.4 Goals

Thus, in order to properly virtualize memory in an efficient and useful manner, we need to take into account three critical goals.

- **Transparency:** The OS should implement virtual memory in a way that is invisible to the running program. The program should behave as if it is addressing physical memory.
- **Efficiency:** Virtualization of memory should be efficient in terms of *time* and *space*. Too much time should not be spent virtualizing and we should not use too much space in order to support virtualization.
- **Protection:** The OS should **protect** processes from one another. Processes should not be able to affect anything outside of their address space. Thus, **isolation** is a key component of memory virtualization.

In the following chapter, we will discuss the **mechanisms** and **policies** necessary to implement memory virtualization.

11 Chapter 14: Interlude: Memory API

11.1 Types of Memory

There are two types of memory that are allocated in C, the **stack** and the **heap**.

For **stack** memory, the memory allocation/deallocation is managed *implicitly* by the compiler. Sometimes it is called **automatic memory**.

For **heap** memory, memory allocation/deallocation must be handled **explicitly** by the user. Usage of the heap takes much more responsibility, and can have more challenging of an interface. Due to these reasons, we will focus on heap memory throughout the remainder of this chapter.

11.2 The *malloc()* Call

The ***malloc()*** call is used to dynamically allocate memory onto the heap. It takes an argument of *size_t*, which describes how many bytes of memory to allocate. It is better practice to use the *sizeof()* operator here in order to specify the correct number of bytes.

> **Note:** the *sizeof()* operator is not a function, but a *compile-time* operator. That is, the value is known at compile time, so the compiler substitutes the operator with a real value before running the program.

11.3 The *free()* Call

The *free()* call is used to free dynamically allocated memory. It takes only a pointer that is returned by *malloc()*. Thus, it it clear that the size of the dynamically allocated region must be tracked by memory.

11.4 Common Errors

Memory management often leads to a number of errors. In fact, many new languages now incorporate support for **automatic memory management**. In such languages, you never have to call anything to free space. Instead, a **garbage collector** runs and sees what memory you no longer have references to and frees up that memory.

Let's take a look at some common mistakes that can be made during dynamic memory allocation.

11.4.1 Forgetting to Allocate Memory

Below is an example of forgetting to allocate memory before assigning to it.

```
{
    char *src = "hello";
    char *temp;             // unallocated!
    strcpy(dst, src);       // will cause segfault!
}
```

Listing 1: Forgetting to call *malloc()*

The code above will cause a **segmentation fault**, as memory was never allocated for the pointer. The correct code is shown below.

```
{
    char *src = "hello";
    char *temp = (char*) malloc(strlen(src) + 1);
    strcpy(dst, src);       // no more errors
}
```

Listing 2: A correct implementation

11.4.2 Not Allocating Enough Memory

The *strlen()* function does not include the **null terminating** byte at the end of the string. Shown below is an example of not allocating enough memory.

```
{
    char *src = "hello";
    char *temp = (char*) malloc(strlen(src));
    strcpy(dst, src);       // no more errors
}
```

Listing 3: Not allocating enough memory

Such a mistake could cause a **buffer overflow**, where the system reads/writes past the end of the allocated space. Below is a fix.

```
{
    char *src = "hello";
    char *temp = (char*) malloc(strlen(src) + 1);
    strcpy(dst, src);       // no more errors
}
```

Listing 4: A correct implementation

11.4.3 Forgetting to Free Allocated Memory

This is known as a **memory leak**. This happens when the user forgets to call *free()* after dynamically allocating memory.

This may not necessarily result in bad behavior for short-lived programs, as the memory will be freed once the process dies. However, long programs such as servers will suffer from memory leaks over time.

11.4.4 Underlying OS Support

Malloc() and *free()* are not system calls. Rather, they are library calls that were built on top of system calls.

One system call that the library uses is *brk*, which is used to change the location of the program's **break**: the location of the end of the heap. A user should never attempt to call *brk*, as it will most likely lead to nasty behavior.

12 Chapter 17: Free Space Management

In this chapter, we will discuss the topic of **free space management**. Managing free space can be easy when space is divided into same-size portions, as is shown through **paging**. However, free space management becomes more complicated when we have variable-size chunks of free space, as is the case when dynamically allocating memory with *malloc()*.

When users dynamically allocate memory, or when an OS uses **segmentation** to manage memory, they are susceptible to **external fragmentation**. That is, the available memory gets split up into different-sized chunks, and in some cases no single chunk can satisfy a memory request, even though there is enough free memory in total to satisfy the request.

12.1 Assumptions

We will make several assumptions throughout the duration of this chapter. The first is that free memory is managed through some kind of **free list**. This data structure, which does not need to be a list, contains references to all free chunks of space in memory.

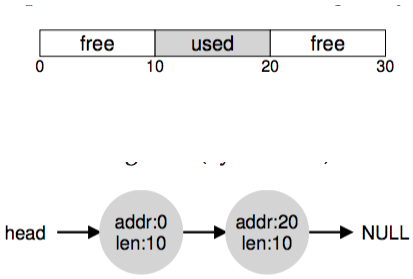
Our second assumption is that we are concerned more about **external fragmentation** than **internal fragmentation**. Internal fragmentation occurs when the OS gives more memory than requested, and there is extra, unasked-for space that is not utilized.

Finally, we will assume that **compaction** is not possible. That is, we cannot reallocate memory once it is allocated for sake of space management. Once a program/process has requested memory, it is allocated until the caller calls *free()*.

12.2 Low Level Mechanisms

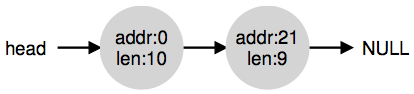
12.2.1 Splitting and Coalescing

Take a look at the following space of memory and its corresponding **free list**.



Note that a memory request of 15 bytes, would *fail* on the memory space shown above. Though there are 20 bytes free, there is no **contigous** portion of 15 bytes free, as shown in the free list.

However, how does a free list change after a request for 1 byte is processed? The allocator will perform a **splitting** operation, where it finds a chunk in the free list large enough, then *splits* it into two chunks: the chunk that gets returned to the caller, and the remainder of the chunk that stays in the free list. For example, the free list above would look like the following after a request for 1 byte.



But what happens in the first example, after the user calls *free()* to return the 10 allocated bytes to memory. If the new chunk was simply added to the free list, as shown below, *there would be problems!*

In the above diagram, there are now 3 10-byte chunks, rather than a single 30-byte chunk even though the available memory is contiguous. Thus, to better handle this scenario, the memory allocator **coalesces** memory by carefully comparing the addresses of newly freed chunks with surrounding addresses in the free list. If the chunks happen to be next to each other, the allocator combines the two chunks into one larger chunk, as shown below.

12.2.2 Tracking the Size of Allocated Regions

The call to *free()* does not require the byte offset to free. Thus, it is clear that the *malloc* library must keep track of the size of allocated memory. This is done through allocating a **header** block of memory just before the real chunk, as shown below.

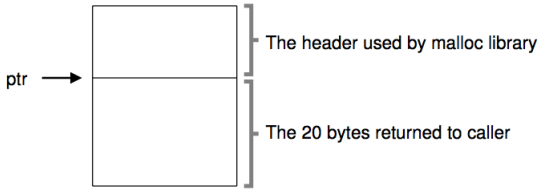


Figure 17.1: **An Allocated Region Plus Header**

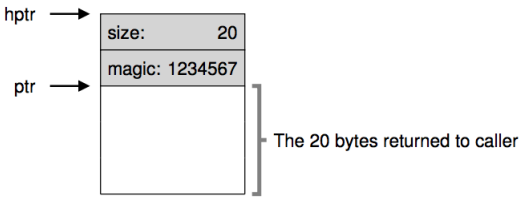


Figure 17.2: **Specific Contents Of The Header**

The header can contain the size of the allocated chunk, as well as a magic number for verification/sanity when freeing. When *free()* is called, the *malloc* library checks the size of the chunk and verifies the magic number.It is important to note that when a call to *malloc()* requesting N bytes is processed, **it really looks for $N + sizeof(header)$ bytes of free space.**

12.2.3 Embedding a Free List

How is a **free list** stored in memory? For obvious reasons, we can't just call *malloc()*. Let's assume we are using a 4096 byte heap and take a look at the following possible implementation.

We can start off by defining our free list node.

```
typedef struct __node_t {
    int size;
    struct __node_t *next;
} node_t;
```

Listing 5: Defining our free list node

Next, we must allocate our 4096 bytes of heap memory and initialize our first free chunk.

```
{
    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;
}
```

Listing 6: Initializing the heap

After this initialization, our heap looks like the following diagram.

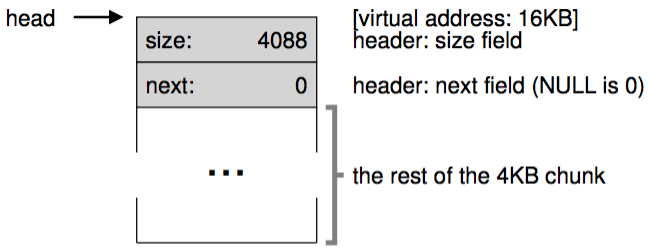


Figure 17.3: **A Heap With One Free Chunk**

We have one free chunk of space, that is of size 4088 (due to the 8 bytes of our header). What happens when we get a request for 100 bytes of space? First, we must find a space that is big enough to handle this 100 byte request (or really 108 bytes due to the header). Because we only have one available chunk, we split this chunk in two, return the newly allocated portion to the user, and keep the second in our list, making the heap as shown below.

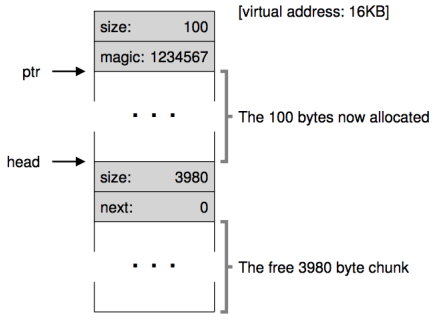


Figure 17.4: **A Heap: After One Allocation**

Now let's assume that we have 3 100-byte allocated regions (not including headers), and we want to *free()* one of them. The allocator frees this memory, then adds it to the free list with the *next* pointer pointing to our large remaining free chunk, as shown below.

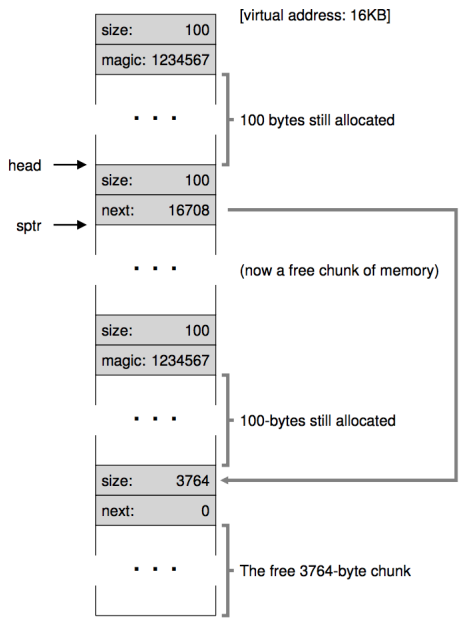


Figure 17.6: **Free Space With Two Chunks Allocated**

When all of the chunks are freed, we need to make sure to **coalesce** them, or we will have severe **external fragmentation**. Such a result is shown below. This is simple, simply go through the list and check for adjacent memory addresses and **merge** them into a single larger chunk.

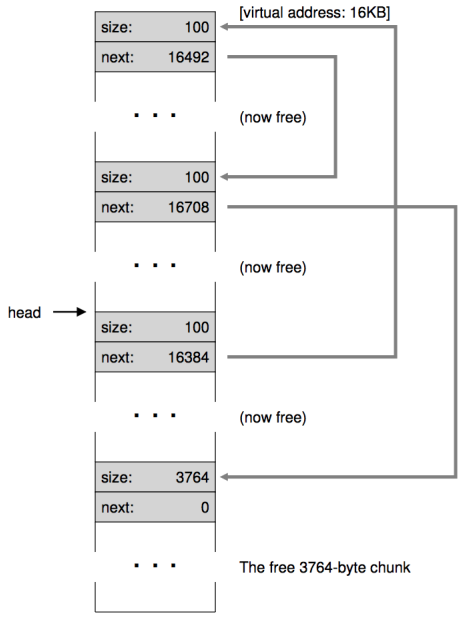


Figure 17.7: **A Non-Coalesced Free List**

12.2.4 Growing the Heap

A rudimentary method of handling a filled heap would be to just fail. However, many memory allocators start with a small heap and reallocate as space is needed, using the *sbrk* system call to grow the heap.

12.3 Basic Strategies

There are several strategies for managing free space.

12.3.1 Best Fit

This approach goes through the entire free list and searches for the *smallest* chunk that is large enough to fulfill the request. It's main downfall is performance, as it must search through the list exhaustively.

12.3.2 Worst Fit

This approach is the opposite of best fit, and aims to keep large chunks of free memory on the heap for later use. It searches for the largest chunk of memory, splits it and keeps the remaining chunk in the free list. It's main disadvantage is performance, but it is helpful in reducing **external fragmentation**, as large blocks of memory are left in the free list.

12.3.3 First Fit

The first fit just searches for the first chunk that is big enough to handle the request. It's main advantage is speed, but pollutes the beginning of the free list with small object. **Address-based ordering** is a good fix for this, as it helps with **coalescing** and reduces **fragmentation**.

12.3.4 Next Fit

Next fit is the same philosophy of first fit, but it keeps track of an extra pointer to where it searched last and searches from there on. It is better in terms of not polluting the beginning of the list and is not hindered by performance.

12.4 Other Approaches

12.4.1 Segregated Lists

Segregated lists are a popular approach to free space management. Essentially, the technique just keeps different lists of different size chunks for memory allocation, which is more optimal for applications that make popular-sized requests. The advantages are **easy coalescion** and **quicker allocation/free requests** due to the lack of a need to search for the "right" chunk. The complication of this implementation is: how much memory should be reserved for each list?

12.4.2 Buddy Allocation

Buddy allocation essentially treats free memory as a big space of 2^n bytes. When a request is made, it splits the memory in 2 over and over again until a block that is *barely* big enough to handle the request is found.

The downfall of this approach is through **internal fragmentation**. Only blocks that are of size 2^n are able to be allocated, commonly leaving unused space in the allocated blocks.

This technique is great for coalescion, as when a block is freed, it checks if its "buddy" is freed, then recursively does this until it coalesces to the largest possible block.

13 Chapter 18: An Introduction to Paging

The first approach to space-management, **segmentation**, where we split space into *variable* size pieces (such as the heap, stack, etc.) has the underlying problem of **fragmentation**.

Thus, a better and more novel approach is to introduce **paging**, where we split memory into *fixed* size units, which we call a **page**. Thus, memory is viewed as an array of fixed-size slots called **page frames**, which can hold a single virtual memory page.

The main focus of this chapter is to **virtualize memory with paging in order to avoid fragmentation.**

13.1 A Simple Example and Overview

Paging is made possible through splitting our memory into fixed-size chunks called **page frames**. Take a look at the following example of paging within a 128 byte physical memory space.

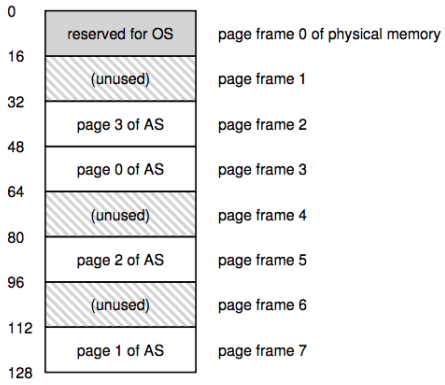


Figure 18.2: **A 64-Byte Address Space In A 128-Byte Physical Memory**

In this program, we have split our 128 byte physical memory into an **address space** of 64 bytes, which take up page frames 2, 3, 5, and 7. But what are the **advantages** of paging?

- **Flexibility:** We no longer need to make underlying assumption about our processes in order to cater to segmentation. A correctly implemented paging approach will support abstraction regardless of how the process uses it.
- **Simplicity:** When we need to create an address space, the OS just needs to find the correct amount of free pages, maybe through a **free list** or such.

In order to keep track of where a *virtual page* lies in *physical memory*, operating systems usually keep a *per-process* data structure called a **page table**. Page tables map virtual page numbers to physical page frames through *address translation*.

Each virtual address can be *mapped* to its physical address through address translation. Virtual addresses can be split up into two components, the upper bits are the **virtual page number (VPN)**, while the lower bits are the **offset** within that page. We can translate the VPN to a **Physical Frame Number (PFN)**, and then index into that page with our offset in mind. Below is a diagram showing this operation.

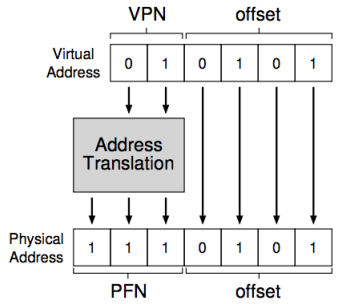


Figure 18.3: **The Address Translation Process**

13.2 Where Are Page Tables Stored?

Page tables can be extremely large. Because of this, we don't store the page tables in hardware, but in memory. We will see later that even page tables can be **virtualized**, but let's assume for now that they reside in physical memory.

13.3 What's Inside the Page Table

A page table consists of **page table entries**. There are multiple ways to structure a page table, but the most straightforward data structure is a **linear page table**, which is just an array.

In order to obtain the PTE, the OS indexes into the page table by VPN. After this, the PTE, stores the PFN with the upper bits (20 for x86 machine), and stores some more information using bit flags for the bottom bits. Information on these common bit flags is shown below.

- **Valid Bit:** tells whether the translation is valid. For example, a process will have the code and heap at one end of the address space, and the stack at the other end. All space in between is **invalid**, and will cause a trap.
- **Protection Bits:** these bit flags tell whether the page can be read, written to, executed, etc.
- **Present Bit:** tells whether the page is in physical memory or disk, in which case it must be retrieved.
- **Dirty Bit:** tells whether the page has been altered since it was brought into memory.
- **Reference Bit:** tells whether the page has been opened/is popular. Useful during **page replacement** (discussed later).

13.4 Paging: Also Too Slow

Paging is also a very slow and extensive process. Simply storing a variable involves an extensive translation process of the virtual address to a physical address. Here is a sample process for translating the address.

```
1 {
2     /* Here we are simply finding the Virtual Page Number to index our Page Table. We need
      the upper bits. */
3     VPN = (Virtual Address & VIRTUAL_ADDRESS_MASK) >> SHIFT
4
5     /* Offset is lower bits of virtual address */
6     offset = Virtual Address & OFFSET_MASK
7
8
9
```

```
    /* Assuming we have a pointer to the beginning of the page table, go to index VPN to
    obtain our PTE */
    PTE = PAGE_TABLE_BASE_POINTER + (VPN * sizeof(PTE))

    /* From the page frame number (upper bits of PTE), our physical address is that PFN
    shifted by SHIFT (due to the way we partitioned our fixed-size page frames) PLUS our
    offset within that page frame (ORing adds in this case due to no overlap of bits). */
    Physical Address = (PTE.PAGE_FRAME_NUMBER << SHIFT ) | offset
}
```

Listing 7: Process for translating address

Thus, just issuing a memory load would be the following code.

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

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

// Fetch the PTE
PTE = AccessMemory(PTEAddr)

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

Figure 18.6: Accessing Memory With Paging

Thus, **paging** is both a **long** and **space-intensive** process. It should be implemented correctly in order for it to have benefit, which will be discussed in the next chapters.

14 Chapter 19: Paging: Faster Translations (TLBs)

In order to make **paging** possible, we need some help from hardware. This is where the **translational-lookaside buffer (TLB)** comes into play. This TLB is simply a hardware cache of popular virtual address to physical address translations. Because of this, the OS doesn't always need to consult the page table; it can just look at the TLB translation and run with it!

14.1 TLB: Basic Algorithm

The basic algorithm for using the TLB is as follows.

When the TLB contains the virtual page translation we're looking for, it's called a **TLB hit**. These are very fast, and enable us to use paging efficiently. However, when the TLB does not have the page translation (a **TLB miss**), there is an *extra* memory load *in addition* to the long page table reference process. Thus, **TLB hits are really fast, but TLB misses are *really* slow**. So how does the TLB make anything better?!

14.2 Temporal and Spatial Locality

The TLB is advantageous due to two concepts: **temporal locality** and **spatial locality**.

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                  // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else if (CanAccess(PTE.ProtectBits) == False)
        RaiseException(PROTECTION_FAULT)
    else
        TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
        RetryInstruction()
```

Figure 19.1: **TLB Control Flow Algorithm**

Spatial locality is exhibited when a program repeatedly references memory address that are close to each other, and will most likely result in successful **TLB hits**. In modern systems, pages can be very large (4 KB) and so they benefit from **spatial locality**.

Temporal locality is exhibited when a program repeatedly references previously accessed memory, so the successive references are all **TLB hits**.

Thus, spatial locality regards a program repeatedly references items close in **space**, while temporal locality regards a program repeatedly references items close in **time**.

14.3 Who Handles a TLB Miss?

In older systems, TLB misses were handled by hardware control flows that were strict and unflexible. The hardware had to keep a **page table base pointer** and know the **exact format** of the page table entries. These **hardware-managed TLBs** were soon phased out by more flexible and simple **software-managed TLBs**.

In a **software-managed TLB**, a **TLB miss** results in the **trap**, which then raises the privilege level to kernel mode and jumps into a **trap handler**. This trap handler executes privileged instructions that look up the page table entry, update the TLB to contain the translation, and retry the TLB check. However, we need to be sure of *two details*.

1. The return from trap instruction must be a little different than usual, as it must return and *retry* the TLB fetching.
2. The OS must ensure *infinite translational-lookaside buffer misses* (in the case that the address of the trap handling code is not within the TLB). This can be alleviated by either:
    - Keeping the trap handling code in physical memory, where it is **unmapped** and not subject to translation
    - Hard wiring several TLB entries for the handler code so that it **never misses those entries**.

The **software-managed TLB** benefits from **flexibility** and **simplicity**. The OS can use any format for the page table without changing hardware and the hardware only has to raise an exception when a **TLB miss** occurs.

14.4 What's Inside the TLB?

A typical **TLB** may have 32, 64, or 128 entries and may be **fully associative**. That simply means the entries may be stored anywhere in the TLB, so to find an entry the hardware will search the entire TLB in parallel. A typical TLB entry looks like the following:

$$VPN \mid PFN \mid other\ bits$$

Below are some examples of the "other bits".

- **valid:** tells whether the TLB entry is a valid translation (useful upon boot up when no TLB entry is valid, not the same as page table valid bit)
- **protection bits:** how a page can be accessed (as in page table)
- **other miscellaneous:** dirty bit, address-space identifier bit, other page table bits

14.5 TLB Issue: Context Switch

What happens when the OS issues a context switch? The TLB's translations are *only relevant to the current process' page table entries*, so a lot of stuff could go wrong if a new process were to reference the TLB. In addition, having multiple processes occupy the TLB doesn't allow the hardware to distinguish between different process' translations. How can this be fixed?

A simple solution is to **flush** the TLB every time we do a context switch. This ensures that there will never be a misreferencing of the TLB's translations. But this is costly, as it causes many TLB misses upon every context switch.

A different approach is to store an **address space identifier** within the TLB entry. This way, multiple processes can use the TLB and it is possible to distinguish each different process' relevant translations!

14.6 TLB Issue: Replacement Policy

When the **TLB** creates another entry, it must **replace** an old one. Such **cache replacement** is an important policy to consider.

One approach is through **least recently used** replacement. It's simple; when making space for a new entry, just evict the one that hasn't been used the longest. This takes advantage of locality, but could fall prey to programs that don't favor such locality.

Another approach is **random** replacement, which is even simpler and can avoid corner cases.

14.7 Summary

With the addition of hardware, virtualization of memory is now much more feasible. The incorporation of the **translational-lookaside buffer** allows us to perform most memory translations without having to look through the page table. It's almost as if we are achieving virtualization without virtualizing (in terms of overhead cost).

The TLB is not omnipotent, however. Programs that address more pages than the TLB contains and exceed the **TLB coverage** will experience many TLB misses. Thus, even though the TLB can be extremely helpful, it is not invincible by any means.

15 Chapter 21: Beyond Physical Memory: Mechanisms

Until now, we have assumed that we had enough memory to hold all of our pages, but that is not true. In order to support many concurrently-running large address spaces, as we must in the **era of multi-programming**, we need to use our **hard disk drive** to store some pages.

15.1 Swap Space

In order to support many running processes, we need to take a small portion of disk and reserve it for **swap space**. Let's assume the OS can read and write pages to/from the swap space, so it must remember a page's **disk address**.

The addition of **swap space** allows for the **impression of a large virtual address space for many process using a small physical address space.** Look at the following example, where we have a 4-page physical memory and an 8-page swap space. 3 processes are running, but only some of their pages are present within memory, giving the impression of a larger address space than they are really using.

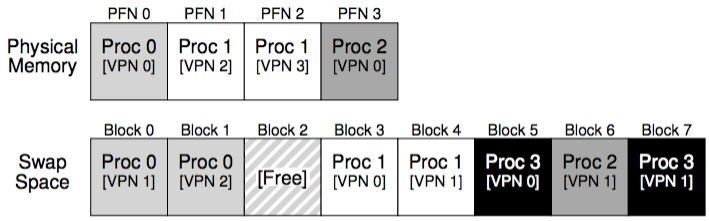


Figure 21.1: **Physical Memory and Swap Space**

15.2 The Present Bit

In order to support our **swap space**, we need to change some higher level machinery to cater to this new system. When a TLB miss occurs, there is a chance that the resulting page table entry (PTE) links to a page that is *not present in memory*. In this case, the **present bit** would be set to 0, and the page is really present on disk. Accessing a page that is not on memory is referred to as a **page fault**.

15.3 The Page Fault

Upon a page fault, the OS invokes a **page fault handler** to service the fault. The OS must swap the page from disk back into memory, but how does it know where it is on disk? In many modern systems, the page table entry is a good place to store the disk address. Because the entry is not valid, the PFN bits can be used to do so.

When the I/O completes, the OS updates the page table entry to be **present** and to have the correct PFN. It then retries the instruction attempting to generate the page.

15.4 What If Memory Is Full?

Until now, we had ignored the fact that memory may be full when attempting to bring in a page. In this case, we must choose a page to be **paged out** to disk. It turns out that making the correct choice here can be the different between several magnitudes of code run time, so these **page replacement policies** will be discussed in depth next chapter.

15.5 Page Fault Control Flow

Putting everything together, we can now show the control flow for a memory access. Upon a **TLB miss**, the hardware checks if the PTE is **valid** and **present**. If not, it issues a page fault. Below is the hardware implementation (for the hardware-managed TLB), which now supports the TLB and page faulting/swapping.

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

Figure 21.2: **Page-Fault Control Flow Algorithm (Hardware)**

Below is the OS software that implements the **page fault handling**.

```
1  PFN = FindFreePhysicalPage()
2  if (PFN == -1)              // no free page found
3      PFN = EvictPage()       // run replacement algorithm
4  DiskRead(PTE.DiskAddr, PFN) // sleep (waiting for I/O)
5  PTE.present = True          // update page table with present
6  PTE.PFN     = PFN           // bit and translation (PFN)
7  RetryInstruction()          // retry instruction
```

Figure 21.3: **Page-Fault Control Flow Algorithm (Software)**

15.6 When Replacement Really Occurs

In reality, the OS doesn't only swap out pages when we run out of physical memory. Most OS' like to keep a small amount of memory free, and do so with a **High Watermark (HF)** and a **Low Watermark (LF)**. When there are less than LW free pages, a **background thread** runs that evicts pages until HW pages available. This thread is called the **swap daemon**.

By performing many replacements at once, new optimizations are possible. Many operating systems swap out pages by **group** or **cluster** in order to make disk access much less intensive by reducing overhead. Such reduction in overhead actually increased performance noticeably!

16 Chapter 22: Beyond Physical Memory: Policies

Virtual memory management sounds convenient when we have infinite memory, but this isn't the case. In reality, OS's must feel **memory pressure**, and must make good decisions of which pages to **page out** in order to swap in more pages.

16.1 Cache Management

Because physical memory only stores a portion of all of our pages, it can be viewed as a **cache** for virtual memory pages. In order to maximize performance, we should **minimize cache misses**. If we know the number of cache misses, we can calculate **average memory access time (AMAT)**.

$$AMAT = T_M + (P_M * T_D)$$

Where T_M is the time to access memory, P_M is the probability of a **cache miss**, and T_D is the time to access disk.

Having a low probability of a cache miss can result in magnitudes of saved memory access time. Thus, it is important to come up with strong policies for paging out, in order to minimize cache misses.

16.2 The Optimal Replacement Policy

The optimal replacement policy was actually figured out; it's simple, but very hard to implement. Essentially, **when evicting a page, choose the page that will be needed farthest in the future**. This actually leads to the best **hit rate** for our cache, but as stated before, it's impossible for the OS to tell the future. Though this policy in unfeasible, it can be good to comparison other policies to the *optimal policy* to gauge its merit.

16.3 A Simple Policy: FIFO

First In, First Out (FIFO) is the first policy of discussion. It is very simple to implement, but has a rather poor hit rate. FIFO simply can't understand the importance of pages, so it will make bad choices in many scenarios.

16.4 Another Simple Policy: Random

Random eviction is another simple policy, but it has similar performance issues. Sometimes random can be close to optimal in its hit rate, but sometimes it can be worse then FIFO - it's all up to the luck of the draw.

16.5 Using History: LRU

FIFO and Random both have the same issue: they may kick out a page that will be referenced soon thereafter. A better policy, **Least Recently Used (LRU)**, takes advantage of **locality** in order to better our hit rate. LRU tends to have a better hit rate, as programs typically adhere to spatial and temporal locality. However, locality is not foolproof, and programs may access memory randomly, in which case LRU could perform badly as well.

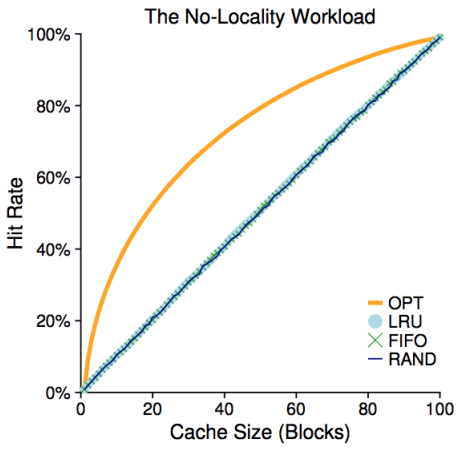
16.6 Workload Examples

Let's take a look at several **workload** scenarios,

16.6.1 No-Locality Workload

Let's say that the program we are running takes no advantage of locality, and accesses 100 *distinct* pages at random through a total of 10k accesses. Let's take a look at how our different policies perform through a series of different caches sizes.

In this example, all of our realistic policies perform about the same, while the optimal policy is obviously the winner. It's important to note that LRU works no better than Random and FIFO when program's don't exhibit locality. The graph is shown below.



16.6.2 80-20 Workload

Let's take a look at the next workload, named the **80-20 workload**. In this example, we'll assume that 80% of our accesses are made to 20% of our pages, and the remaining 20% of our accesses are made to 80% of the pages.

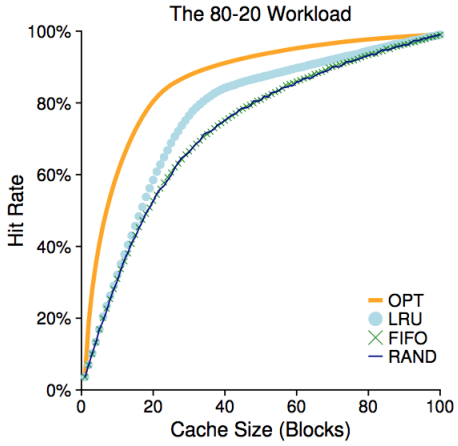


Figure 22.7: **The 80-20 Workload**

In this graph, we see that LRU does perform better than FIFO and Random, but only by a little bit. Whether this slight boost in hit rate will make a large different or not is dependent on the penalty for cache misses. If a miss is very costly, then even a slight improvement in hit rate can make a large difference!

16.6.3 The Looping-Sequential Workload

Our final workload we will consider is the **looping-sequential workload**. In this example, we have 50 distinct pages and we loop through them in order for 10,000 accesses. How do each of our policies fair here?

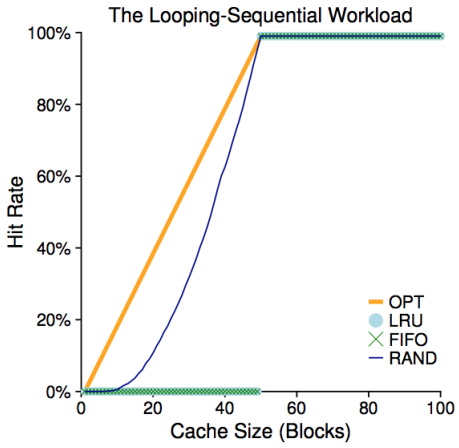


Figure 22.8: **The Looping Workload**

Surprisingly, our Random policy works very well, and our LRU/FIFO policies perform terribly for cache sizes less than 50. It follows than one advantage of the Random policy is that it does not fall prey to edge cases such as this.

16.7 Implementing Historical Algorithms

Implementing the LRU algorithm can result in costly extra instructions that can drastically reduce its feasibility. For LRU, we need to keep some data structure and adjust it on *every page access*, moving the page to the front of the list. For FIFO, we only need to perform extra operations upon *page faults*. In addition, LRU must scan the entirety of the page list to find which page was the least recently used. But the good news is, we don't really need to find the *least* recently used page - we can approximate and improve our run time!

16.8 Approximating LRU

We can approximate LRU through the use of some hardware. In particular, we can keep a **use bit** (or **reference bit**) to help us out. Whenever a page is referenced, it's use bit is set to 1. Changing this bit to 0 is the responsibility of the OS.

In order to approximate LRU using the use bit, the OS can use a multitude of approaches. The most simple, however, is the **clock algorithm**. Essentially, the OS keeps a circular list of the pages along with a pointer (almost like a clock hand) to a page (doesn't matter which), and when a page must be evicted, it checks if the **use bit** is 1. If it is, then the OS sees that the page has been referenced recently and sets the bit to 0. Then, it advances the pointer and repeats this process until it finds a page with a use bit set to 0, where it evicts this page. Below is a graph of 80-20 hit-rate including our clock algorithm.

16.9 Considering Dirty Pages

When a page has been written to - in which case its **dirty bit** has been set - it *can't just be thrown away upon eviction*. It must be written to disk in order to save the changes. This is a very costly operation, and so VM systems prefer to only evict clean pages.

In order to implement such a preference, the VM system can simply check for a page with a **use bit** of 0 and a **dirty bit** of 0.

16.10 Other VM Policies

The OS also must decide when to bring a page *in* to our cache. Most page selections involve **demand paging**, where a page is brought into memory "on demand", or when it is accessed.

Another policy is how the OS writes pages to disk. This is an expensive operation, and so many OS's write to disk in **groups** or **clusters**, performing a single large write to minimize overhead.

16.11 Thrashing

When memory is oversubscribed, and the memory demands of our running processes exceeds our physical memory, **thrashing** occurs. When a system thrashes, it is constantly paging, which is *extremely* slow.

There are several ways to cope with thrashing. One approach is to simply reduce the number of processes running, in order to reduce the overall **working set** (the number of pages the processes use actively). This is called **admission control**.

Another approach is to run an **out-of-memory killer**, which is a thread that runs when memory is oversubscribed whose job is to kill a memory-intensive process. Though this is successful at alleviating memory pressure, it is quite draconian, and can have problems. For example, if it happens to kill a server and renders other applications useless.

17 Chapter 25: A Dialogue On Concurrency

Omitted due to breadth.

18 Chapter 26: Concurrency: An Introduction

A **multi-threaded program** has multiple points of execution. In programs that run multiple **threads**, the different threads *almost behave like different processes*. They have their own program counters, their own set of registers, but the main difference here is that **threads share the same address space**.

In order to switch execution between threads, we still need to perform a **context switch**. However, instead of saving the thread state to a **process control block (PCB)**, as we did for processes, we now need **thread control blocks (TCBs)**. In addition, the context switch is a bit different in the fact that we no longer need to change our page tables, because our threads reference the same **address space**.

Another major difference of multi-threaded programs is the presence of **several stacks**. Because each thread runs independently, each needs its own stack, which we sometimes call **thread-local storage**.

18.1 Why Use Threads?

Threads are useful for two main reasons.

- **Parallelization:** using threads, we can take advantage of systems with multiple processors. Instead of only using a singly CPU for our program, we can have each processor do some work, greatly speeding up our program execution.
- **I/O Overlap:** when a program performs many I/O operations, it must wait for the operation to complete before executing. When using threads, we can still make use of the CPU even while waiting for a response, allowing for **overlap**, and better optimization.

This seems very similar to our previous discussions on *processes*. The main difference is that threads share the same address space, so it makes more sense to use threads for tasks that involve the sharing of data.

18.2 Thread Creation Example

Take a look at the following code showing the creation of two threads.

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

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

The *pthread_create()* function creates and executes our thread, and the *pthread_join()* function waits for a thread to finish execution. When creating the threads we pass in a reference to the thread variable, along with the thread functions and the arguments to each respective function. In this case, the function simply prints out the thread label, but what is interesting is the *order of execution*. **The above program may print A or B first**, depending on how the **scheduler** chooses to schedule the threads. Below are a couple sample control flows of how the scheduler might execute the program.

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns
prints "main: end"		

Figure 26.3: **Thread Trace (1)**

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Figure 26.4: **Thread Trace (2)**

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Figure 26.5: **Thread Trace (3)**

18.3 Why It Gets Worse: Shared Data

We have seen how threads have no **deterministic** order of execution, and this can be a problem when threads reference **shared data**. Let's take a look at a multi-threaded program that references a shared global variable in order to see where things can get complicated.

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include "mythreads.h"
4
5  static volatile int counter = 0;
6
7  //
8  // mythread()
9  //
10 // Simply adds 1 to counter repeatedly, in a loop
11 // No, this is not how you would add 10,000,000 to
12 // a counter, but it shows the problem nicely.
13 //
14 void *
15 mythread(void *arg)
16 {
17     printf("%s: begin\n", (char *) arg);
18     int i;
19     for (i = 0; i < 1e7; i++) {
20         counter = counter + 1;
21     }
22     printf("%s: done\n", (char *) arg);
23     return NULL;
24 }
25
26 //
27 // main()
28 //
29 // Just launches two threads (pthread_create)
30 // and then waits for them (pthread_join)
31 //
32 int
33 main(int argc, char *argv[])
34 {
35     pthread_t p1, p2;
36     printf("main: begin (counter = %d)\n", counter);
37     Pthread_create(&p1, NULL, mythread, "A");
38     Pthread_create(&p2, NULL, mythread, "B");
39
40     // join waits for the threads to finish
41     Pthread_join(p1, NULL);
42     Pthread_join(p2, NULL);
43     printf("main: done with both (counter = %d)\n", counter);
44     return 0;
45 }
```

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

In this example, the job of each thread is to increment our global counter by 10,000,000 through a loop. Our expected output from this program is 20,000,000, but this isn't always the case. In fact, when running the program several times, not only do we not always have an output of 20,000,000, but the output can be *different each time*. This lack of **deterministic** behavior is something very unexpected in the realm of computers.

18.4 The Heart of the Problem: Uncontrolled Scheduling

The reason behind this lack of determinism is due to the **scheduler's** untimely context switches. When incrementing the counter, the machine must perform the following instructions.

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

What happens when the scheduler decides to perform a context switch in between one of these instructions? Bad things! Take a look below at the effects of a badly timed context switch.

			(after instruction)		
OS	**Thread 1**	**Thread 2**	**PC**	**%eax**	**counter**
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	**50**	50
	add $0x1, %eax		108	**51**	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	**50**	50
		add $0x1, %eax	108	**51**	50
		mov %eax, 0x8049a1c	113	51	**51**
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

Figure 26.7: **The Problem: Up Close and Personal**

In this example, one thread increments its register, but a context switch occurs before it stores the register back into the global variable. When the second thread finishes execution and increments the counter, *the old thread is not aware of the new global counter value, and so it ends storing the same value back in the counter*. A scenario like this, where the output/results of a program depend on the timing execution of the code, is called a **race condition**. Race conditions result in **indeterminate** output, where the output is not **deterministic** and can vary across runs.

When the execution of a segment of code by multiple threads can result in a race condition, it is called a **critical section.** In order to deal with this, we must have some form of **mutual exclusion**, where only one thread can enter the critical section at a time.

18.5 The Wish for Atomicity

In the previous example, if instead of having three separate instructions for incrementing our counter, everything was done **atomically** - that is, all at once - there would be no issues. However, this is simply not a feasible desire, especially for more complex instruction sets such as updating an entire tree.

Thus, we once again need assistance from hardware in order to develop **synchronization primitives**, which will allow us to multi-thread code in a synchronized and controlled manner... with help from the OS of course.

19 Chapter 27: Thread API

19.1 Thread Creation

The API call for creating a thread is as follows.

```
#include <pthread.h>
int
pthread_create(      pthread_t *        thread,
               const pthread_attr_t *   attr,
                     void *             (*start_routine)(void*),
                     void *             arg);
```

This looks more confusing than it really is. Let's go through each of the function parameters.

The first parameter is simply a pointer to a thread struct variable. This initializes the variable, so we can use it to interact with the thread.

The second parameter is the attributes the thread has, which could be used to set the stack size or priority of the thread. This can be NULL most of the time.

The third parameter is a **function pointer**. A function pointer is simply the name of a function, but in our case the function must have several specifications. It must only accept a single argument of type ** void*, and it must return a value of type ** void*.

The final parameter contains the argument that you are passing to the function named by the **function pointer**. Having all of these be **void pointers** ensures that we can pass *any value* to the function, and it can return any value as well.

19.2 Thread Completion

In order to wait for a thread's completion, we call *pthread_join()*. The API call for this function is a little simpler.

The first argument it accepts is a reference to the thread of concern. The second argument is a **void pointer** to the expected return value of our thread function. It is void due to the fact that our function can return any type, and this argument *must* be passed in as a pointer, as the *pthread_join()* function may change the value during its execution. Below is an example program that makes use of *pthread_join()*.

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <assert.h>
4  #include <stdlib.h>
5
6  typedef struct __myarg_t {
7      int a;
8      int b;
9  } myarg_t;
10
11 typedef struct __myret_t {
12     int x;
13     int y;
14 } myret_t;
15
16 void *mythread(void *arg) {
17     myarg_t *m = (myarg_t *) arg;
18     printf("%d %d\n", m->a, m->b);
19     myret_t *r = Malloc(sizeof(myret_t));
20     r->x = 1;
21     r->y = 2;
22     return (void *) r;
23 }
24
25 int
26 main(int argc, char *argv[]) {
27     pthread_t p;
28     myret_t *m;
29
30     myarg_t args = {10, 20};
31     Pthread_create(&p, NULL, mythread, &args);
32     Pthread_join(p, (void **) &m);
33     printf("returned %d %d\n", m->x, m->y);
34     free(m);
35     return 0;
36 }
```

Figure 27.2: **Waiting for Thread Completion**

Note: It is important that the thread function does not return any stack-allocated variables, as when the thread completes, these variables go out of scope, which is super bad.

19.3 Locks

In order for us to have **mutual exclusion** within **critical sections**, we must use the thread API to implements **locks**. Let's take a look at the functions.

```
{
    int pthread_mutex_lock(pthread_mutex_t *mutex);
    int pthread_mutex_unlock(pthread_mutex_t *mutex);
}
```

Listing 8: Lock related functions

We must place these functions strategically around our **critical section**, in order to implement our mutual exclusion. This is shown below. **When a thread holds the lock, no other thread can enter the critical section until the lock is released.**

```
{
    pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
    int pthread_mutex_lock(pthread_mutex_t *mutex);
    x++;
    int pthread_mutex_unlock(pthread_mutex_t *mutex);
}
```

Listing 9: Using a lock around a critical section

It is **crucial** to initialize the lock, however. This can be done in two different ways. The first is with the PTHREAD_MUTEX_INITIALIZER, shown above. You can also use the *pthread_mutex_init* function.

Note: the above code does not check for error codes, as it should. In order to be sure that the code is running correctly, a wrapper function could be used to check the return code of the calls.

19.4 Condition Variables

Another way threads can communicate is through **condition variables**. In some cases, we want a thread to wait for a signal from another thread before executing past a certain point. We use two functions to implement such condition variables, shown below.

```
{
    int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
    int pthread_cond_signal(pthread_cond_t *cond);
}
```

Listing 10: Condition variable functions

In order for one thread to wait for another thread's signal, the following usage may be useful.

```
{
    pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
    pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
    Pthread_mutex_lock(&lock);
    while (ready == 0)
        Pthread_cond_wait(&cond, &lock);
    Pthread_mutex_unlock(&lock);
}
```

Listing 11: Usage of condition variables (waiting thread)

On the other end, the usage for waking this thread would be as follows.

```
{
    Pthread_mutex_lock(&lock);
    ready = 1;
    Pthread_cond_signal(&cond);
    Pthread_mutex_unlock(&lock);
}
```

Listing 12: Usage of condition variables (signaling thread)

There are a few things to note here. First, we must make sure to use locks in this implementation in order to eliminate the possibilities of **race conditions** inside the **critical section** of setting the ready flag.

Second, the *Pthread_cond_wait* function accepts a reference to a lock. This is *crucial*, as the function actually releases the lock. If it didn't, then the other thread would never be able to hold the lock and release the first thread!

19.5 Summary

The thread API has some particularities, but it is simple when explained. The real complicated aspects of threads, however, lie in the tricky logic to building concurrent programs rather than the thread API. This will be discussed further in the following chapters.

20 Locks

20.1 Locks: The Basic Idea

We discussed **locks** in the previous chapter, but let's go more in depth. A lock is simply a variable (in our case it is a **mutex**) that holds the state of the lock. The lock can either be **available**, where no thread holds the lock, or **acquired**, where a **single** thread holds the lock and is presumably executing a **critical section**.

The *lock()* and *unlock()* semantics are simple. When a thread calls *lock()*, it attempts to acquire the lock. If it is already held by another thread, the lock simple **does not return** while another thread holds the lock. If it is not currently held by another thread, this thread acquires the lock and is able to enter the critical section.

The owner of the lock can call the *unlock()* function, which releases the lock and puts it in an **available** state. From here, if another thread aims to acquire the lock, it is now possible.

Locks allow us to implement **mutual exclusion**, as only a single thread can enter a critical section. It is almost as if the critical section is occurring **atomically**.

20.2 Pthread locks

Pthread locks are called **mutexes**, as they implement this necessary **mutual exclusion**. The Pthread *lock()* and *unlock()* routines also accept a reference to a mutex variable, as we can use *different lock variables for different critical sections*. This allows for increased concurrency. Below is some sample code for implementing a Pthread lock.

```
1 {
2     pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
3
4     Pthread_mutex_lock(&lock); // wrapper for pthread_mutex_lock()
5     balance = balance + 1;
6     Pthread_mutex_unlock(&lock);
7 }
```

Listing 13: A Pthread lock example

20.3 Building/Evaluating Locks

In building an efficient lock, we need to address several goals.

- It needs to provide **mutual exclusion** in order to make sure multiple threads don't concurrently enter a critical section.
- It must be **fair**. More specifically, threads mustn't **starve**, and thus never acquire the lock.
- It must have good **performance**; that is, it can't come with too much overhead. There are several different scenarios to evaluate here.
  - Running a single thread
  - Running multiple threads on a single CPU
  - Running multiple threads on multiple CPUs

20.4 Controlling Interrupts

One early method of providing **mutual exclusion** on single-CPU systems was through **disabling interrupts** when entering a critical section.

```
1
2   void lock() {
3       DisableInterrupts();
4   }
5   void unlock() {
6       EnableInterrupts();
7   }
```

Listing 14: *lock()* and *unlock()* routines for controlling interrupts

This approach is very simple and easy to implement, but it has many downfalls.

- This approach allows the program to perform **privileged operations**. A program can now disable interrupts, so the OS must **trust** the program to not misues this privilege, as it could cause many problems.
- This approach **does not work on multiprocessors**. Even if interrupts are disabled, another thread running on another processor could still enter the critical section.
- Turning off interrupts for extended periods of time **leads to interrupts being lost**. This can lead to problems in the system - imagine if there was an interrupt to notify the completion of an I/O operation, but it was ignored. The process that performed the I/O would never be woken up!
- It is slow!

Though this approach seems unfeasible for programs, some OS's still use this form of mutual exclusion to provide **atomicity** to their internal data structures. This makes more sense, as obviously the OS trusts itself to perform privileged operations, so our trust issues disappear.

20.5 A Failed Attempt: Just Using Load/Stores

A simple attempt to creating a lock would be to simply use a **flag** to mark possession of the lock. Take a look at the following implementation of such a lock.

```
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**

In this implementation, the *unlock()* function simply sets our flag to 0, indicating it is available. The *lock()* function checks if the flag is 1, in which case the lock is already held. If this is the case, the thread **spin-waits** until the lock is free.

This implementation has two problems: **correctness** and **performance**. To see why the code is incorrect, let's look at the following scheduler trace.

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**

We see that the *lock()* routine is actually a **critical section itself**. A thread could exit the loop, but get interrupted before setting the flag. The next thread runs the same routine and set the flag to 1, and then the previous thread finishes and sets the flag to 1 too!

20.6 Building Working Spin Locks with Test-And-Set

Because both of our previous approaches are not favorable, we need some help from our old pal: **hardware**. Specifically, we need a single test-and-set instruction that behaves in the following way.

```
1
2 int test-and-set(int *old_ptr, int new) {
3     int old = *old_ptr; // fetch old value at old_ptr
4     *old_ptr = new; // store    new    into old_ptr
5     return old; // return the old value
6 }
```

Listing 15: test-and-set instruction in C code

The key to this implementation is that the test-and-set instruction is **atomic**. All of the operations happen at once, so there is no more need to fear untimely interrupts. With this new hardware, we can build a simple **spin lock**, shown below.

Now that the **testing** of the flag and the **setting** of the flag are **a single atomic operation**, we can ensure **mutual exclusion**! There is one obvious thing to note here, however. Spin locks **only work with**

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available, 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Figure 28.3: **A Simple Spin Lock Using Test-and-set**

preemptive schedulers - those that periodically interrupt a thread to run another. If a spin lock existed on a non-preemptive scheduler, a thread would just spin forever.

20.7 Evaluating Spin Locks

How do spin locks do on correctness, fairness, and performance?

- **Correctness:** now that there is no chance of untimely interrupts, this implementation is **correct**.
- **Fairness:** unfortunately, spin locks are not **fair**. A thread may theoretically spin forever, as the thread that obtains the lock next is pseudo-random.
- **Performance:** for performance, we need to consider two different cases:
  - **for a single CPU machine, spin locks are very slow**. Imagine if the thread that holds the lock was interrupted during its execution of the **critical section**. It would have to wait for all the other $N - 1$ threads to spin-wait for their CPU cycles!
  - **When we have roughly as many CPUs as threads, spin locks are relatively efficient.** The thread that is in the critical section will most likely finish quickly, so a thread spinning on another processor doesn't waste too many CPU cycles.

20.8 Compare-and-Swap

Another hardware implementation that aims to solve mutual exclusivity is **compare-and-swap**. The compare-and-swap instruction works in a similar way to test-and-set, but what matters is that it is **atomic**. Take a look at its C-equivalent implementation below.

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}
```

Listing 16: CompareAndSwap instruction in C code

We can build a **spin lock** with compare-and-swap in the same way we built the previous, and thus the behavior is exactly the same. The new *lock()* code would be the following.

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

Listing 17: compare-and-swap instruction in C code

20.9 Load-Linked and Store-Conditional

Some platforms provide a pair of instructions that work together to help wrap critical sections with mutual exclusivity. One example is the pair of **load-linked** and **store-conditional**, whose C pseud-ocode implementation is below.

```
1
2  int LoadLinked(int *ptr) {
3      return *ptr;
4  }
5
6  int StoreConditional(int *ptr, int value) {
7      if (no one has updated *ptr since the LoadLinked to this address) {
8          *ptr = value;
9          return 1; // success!
10     } else {
11         return 0; // failed to update
12     }
13 }
```

Listing 18: Load-linked and store-conditional

20.10 Fetch-And-Add

One last hardware primitive is **fetch-and-add**. This instruction simply fetches a value and increments the pointer value from its parameter. Look at its C translation below.

```
1
2  int FetchAndAdd(int *ptr) {
3      int old = *ptr;
4      *ptr = old + 1;
5      return old;
6  }
```

Listing 19: fetch-and-add instruction

With this instruction, we can build a **ticket lock**. Using a lock struct with a turn variable, we can assign tickets to threads and execute them in order.

```
1
2  typedef struct __lock_t {
3      int ticket;
4      int turn;
5  } lock_t;
6
7  void lock_init(lock_t *lock) {
8      lock->ticket = 0;
9      lock->turn = 0;
10 }
11
12 void lock(lock_t *lock) {
13     int myturn = FetchAndAdd(&lock->ticket);
14     while (lock->turn != myturn)
15         ; // spin
16 }
17
18 void unlock(lock_t *lock) {
19     lock->turn = lock->turn + 1;
20 }
```

Listing 20: A ticket lock implementation

Essentially, the *lock()* routine grabs a value from a global counter, then increments it. It then waits for the lock turn variable to reach its **ticket** value, where it executes the **critical section**. The **unlock()** routine increments the turn value to be used by the next ticketed thread. In this kind of lock, we have much more **fairness**. Once a thread gets assigned its ticket number, it is guaranteed to execute at some point.

20.11 Too Much Spinning: What Now?

These simple hardware-based locks have one downfall. There's way too much spinning. Imagine running two threads on a single CPU. When the lock-holding thread is interrupted, the next thread is continuously checking a value *that is never going to change in its time slice.* This gets even worse as we have more threads, as we waste $N-1$ time slices! That sucks.

20.12 A Simple Approach: Yielding

Instead of spinning, the thread can just forfeit the CPU. This is possible through the *yield()* system call. This call essentially just moves the thread's state from **running** to **ready**, essentially **descheduling** itself. Take a look at a lock implemention below.

```
1
2  void init() {
3      flag = 0;
4  }
5
6  void lock() {
7      while (TestAndSet(&flag, 1) == 1)
8          yield(); // give up the CPU
9  }
10
11 void unlock() {
12     flag = 0;
13 }
```

Listing 21: Yielding instead of spinning

Unfortunately, the performance of this still isn't great. Though we're not wasting $N-1$ time slices, if the lock-holding thread gets **preempted** in the critical section, the other $N-1$ threads will all just yield in some pattern until it gets back to the lock holding thread. In addition, **starvation** is still an issue; a thread could get caught in an endless yield loop, while other threads repeatedly obtain the lock.

20.13 Using Queues: Sleeping Instead of Spinning

Most of our lock implementations thus far were **very subject to randomness**. In order to combat this, we can keep a queue of threads, and force a thread to sleep when it doesn't have a lock after adding it to the queue. Take a look at the implementation below.

```
1
2  typedef struct __lock_t {
3      int flag;
4      int guard;
5      queue_t *q;
6  } lock_t;
7
8  void lock_init(lock_t *m) {
9      m->flag = 0;
10     m->guard = 0;
11     queue_init(m->q);
12 }
13
14 void lock(lock_t *m) {
15     while (TestAndSet(&m->guard, 1) == 1)
16         ; //acquire guard lock by spinning
17     if (m->flag == 0) {
18         m->flag = 1; // lock is acquired
19         m->guard = 0;
20     } else {
21         queue_add(m->q, gettid());
22         m->guard = 0;
23         park();
24     }
25 }
```

```
26
27 void unlock(lock_t *m) {
28     while (TestAndSet(&m->guard, 1) == 1)
29         ; //acquire guard lock by spinning
30     if (queue_empty(m->q))
31         m->flag = 0; // let go of lock; no one wants it
32     else
33         unpark(queue_remove(m->q)); // hold lock (for next thread!)
34     m->guard = 0;
35 }
```

Listing 22: A ticket lock implementation

Notice that the flag does not get set to 0 when unlocking. This is because when a thread is awoken with *unpark()*, it begins execution as if it returned from *park()*.

Also, there is a race condition in the above solution, right before the call to *park()*. If a context switch were to take place *right* before the call to *park()*, then when the thread gets activated again, it will park without adding itself back to the queue, never executing. This is a **wakeup/waiting race**.

21 Lock-based Concurrent Data Structures

Adding **locks** to data structures makes them **thread-safe**. The implementation of the locks, however, can have effects on performance and correctness of the data structures.

21.1 Concurrent Counters

A counter is a simple data structure, but how do we make it **thread-safe**? Take a look at the implementation below.

```
1  typedef struct __counter_t {
2      int value;
3  } counter_t;
4
5  void init(counter_t *c) {
6      c->value = 0;
7  }
8
9  void increment(counter_t *c) {
10     c->value++;
11 }
12
13 void decrement(counter_t *c) {
14     c->value--;
15 }
16
17 int get(counter_t *c) {
18     return c->value;
19 }
```

Figure 29.1: **A Counter Without Locks**

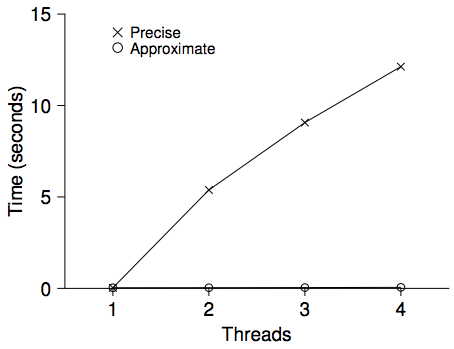
```
1  typedef struct __counter_t {
2      int             value;
3      pthread_mutex_t lock;
4  } counter_t;
5
6  void init(counter_t *c) {
7      c->value = 0;
8      Pthread_mutex_init(&c->lock, NULL);
9  }
10
11 void increment(counter_t *c) {
12     Pthread_mutex_lock(&c->lock);
13     c->value++;
14     Pthread_mutex_unlock(&c->lock);
15 }
16
17 void decrement(counter_t *c) {
18     Pthread_mutex_lock(&c->lock);
19     c->value--;
20     Pthread_mutex_unlock(&c->lock);
21 }
22
23 int get(counter_t *c) {
24     Pthread_mutex_lock(&c->lock);
25     int rc = c->value;
26     Pthread_mutex_unlock(&c->lock);
27     return rc;
28 }
```

Figure 29.2: **A Counter With Locks**

21.1.1 Simple But Not Scalable

This data structure works very well, but unfortunately it is not very good in terms of **scale**. Say we run a function that counts to 1 million, and we multithread to have *each* thread count to a million on its own processor. Take a look at the following graph of how the time scales with more threads.

Ideally, we would want multiple threads running on their own processor to perform as fast as a single thread on a single processor. This is called **perfect scaling**, where our time taken to perform a task does not increase at all as we incorporate more threads, but more work is being done. This is possible through executing the threads on multiple CPUs.



21.1.2 Simple But Scalable

In order to make counters more **scalable**, we can use the **approximate counter**. Approximate counters work by having **multiple** local counters, one per CPU core, and having a single global counter. In addition, there is a different lock for each of the counters. When a thread wants to increment the count, it updates its CPU's counter. When the local counter surpasses a certain threshold, it adds the local counter to the global count and resets the local count.

Because we have a lock for each CPU, threads across different CPUs don't need to compete to count. Thus, **approximate counting is a lot more scalable with locks. How often the local-to-global transfer takes place is dependent on the threshold size.** A larger size is more scalable, but also less accurate as the global counter is updated less frequently. On the other hand, too low of a threshold would give the same performance problems as non-approximate counting. Take a look at an implementation of an approximate counter below.

```
1  typedef struct __counter_t {
2      int             global;           // global count
3      pthread_mutex_t glock;            // global lock
4      int             local[NUMCPUS];   // local count (per cpu)
5      pthread_mutex_t llock[NUMCPUS];   // ... and locks
6      int             threshold;        // update frequency
7  } counter_t;
8
9  // init: record threshold, init locks, init values
10 //       of all local counts and global count
11 void init(counter_t *c, int threshold) {
12     c->threshold = threshold;
13     c->global = 0;
14     pthread_mutex_init(&c->glock, NULL);
15     int i;
16     for (i = 0; i < NUMCPUS; i++) {
17         c->local[i] = 0;
18         pthread_mutex_init(&c->llock[i], NULL);
19     }
20 }
21
22 // update: usually, just grab local lock and update local amount
23 //         once local count has risen by 'threshold', grab global
24 //         lock and transfer local values to it
25 void update(counter_t *c, int threadID, int amt) {
26     int cpu = threadID % NUMCPUS;
27     pthread_mutex_lock(&c->llock[cpu]);
28     c->local[cpu] += amt;                 // assumes amt > 0
29     if (c->local[cpu] >= c->threshold) {  // transfer to global
30         pthread_mutex_lock(&c->glock);
31         c->global += c->local[cpu];
32         pthread_mutex_unlock(&c->glock);
33         c->local[cpu] = 0;
34     }
35     pthread_mutex_unlock(&c->llock[cpu]);
36 }
37
38 // get: just return global amount (which may not be perfect)
39 int get(counter_t *c) {
40     pthread_mutex_lock(&c->glock);
41     int val = c->global;
42     pthread_mutex_unlock(&c->glock);
43     return val; // only approximate!
44 }
```

Figure 29.5: **Approximate Counter Implementation**

21.2 Concurrent Linked Lists

Take a look at the following concurrent linked list. Only the insert function is provided, as all of the others can follow similar practices.

```
1  // basic node structure
2  typedef struct __node_t {
3      int                 key;
4      struct __node_t         *next;
5  } node_t;
6
7  // basic list structure (one used per list)
8  typedef struct __list_t {
9      node_t                *head;
10     pthread_mutex_t     lock;
11 } list_t;
12
13 void List_Init(list_t *L) {
14     L->head = NULL;
15     pthread_mutex_init(&L->lock, NULL);
16 }
17
18 int List_Insert(list_t *L, int key) {
19     pthread_mutex_lock(&L->lock);
20     node_t *new = malloc(sizeof(node_t));
21     if (new == NULL) {
22         perror("malloc");
23         pthread_mutex_unlock(&L->lock);
24         return -1; // fail
25     }
26     new->key  = key;
27     new->next = L->head;
28     L->head   = new;
29     pthread_mutex_unlock(&L->lock);
30     return 0; // success
31 }
32
33 int List_Lookup(list_t *L, int key) {
34     pthread_mutex_lock(&L->lock);
35     node_t *curr = L->head;
36     while (curr) {
37         if (curr->key == key) {
38             pthread_mutex_unlock(&L->lock);
39             return 0; // success
40         }
41         curr = curr->next;
42     }
43     pthread_mutex_unlock(&L->lock);
44     return -1; // failure
45 }
```

Figure 29.7: **Concurrent Linked List**

Notice that *malloc()* is not surrounded by a lock. Assuming it is **thread-safe**, we don't need a lock around it. Factoring this out of our critical section removes the need for a possible extraneous *unlock()* operation in the case that *malloc()* fails. This is very useful, as a very large fraction of bugs result from such exceptional control paths.

Unfortunately, this implementation does not perform well at scale either. One way of improving our concurrency is through **hand-over-hand locking**, where each element of the linked list has its own lock, and when we traverse the list, we grab the next node's lock and give up our current node's. **This permits a high degree of concurrency**, as there is much less time spent waiting for the lock. **However, the overhead of such a list can outweigh its concurrency benefits**, so it is hard to implement in practice.

21.3 Concurrent Queues

We can make any data structure **thread-safe** with a big lock, but this is inefficient. Let's take a look at a better locking system within queues that **makes use of two different locks: one for the head, and one for the tail**.

```
1  typedef struct __node_t {
2      int                 value;
3      struct __node_t    *next;
4  } node_t;
5
6  typedef struct __queue_t {
7      node_t             *head;
8      node_t             *tail;
9      pthread_mutex_t     headLock;
10     pthread_mutex_t     tailLock;
11 } queue_t;
12
13 void Queue_Init(queue_t *q) {
14     node_t *tmp = malloc(sizeof(node_t));
15     tmp->next = NULL;
16     q->head = q->tail = tmp;
17     pthread_mutex_init(&q->headLock, NULL);
18     pthread_mutex_init(&q->tailLock, NULL);
19 }
20
21 void Queue_Enqueue(queue_t *q, int value) {
22     node_t *tmp = malloc(sizeof(node_t));
23     assert(tmp != NULL);
24     tmp->value = value;
25     tmp->next  = NULL;
26
27     pthread_mutex_lock(&q->tailLock);
28     q->tail->next = tmp;
29     q->tail = tmp;
30     pthread_mutex_unlock(&q->tailLock);
31 }
32
33 int Queue_Dequeue(queue_t *q, int *value) {
34     pthread_mutex_lock(&q->headLock);
35     node_t *tmp = q->head;
36     node_t *newHead = tmp->next;
37     if (newHead == NULL) {
38         pthread_mutex_unlock(&q->headLock);
39         return -1; // queue was empty
40     }
41     *value = newHead->value;
42     q->head = newHead;
43     pthread_mutex_unlock(&q->headLock);
44     free(tmp);
45     return 0;
46 }
```

Figure 29.9: **Michael and Scott Concurrent Queue**

As you can see, this concurrent queue uses a head lock for dequeuing, and a tail lock for queuing. **By using two different locks, we allow for much more concurrency**, as a thread calling *dequeue()* doesn't have to wait for a thread calling *queue()* to give up the lock.

21.4 Concurrent Hash Table

We can design a concurrent hash table that actually works very well using our concurrent list from before. Because we have a lock for every list, we have a high degree of concurrency. Take a look at how it compares to a normal concurrent list.

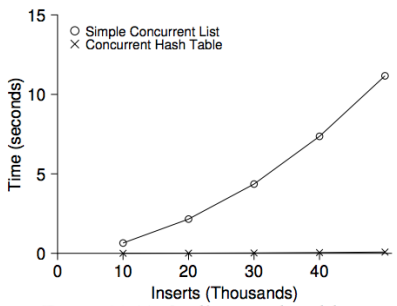


Figure 29.11: **Scaling Hash Tables**

Take a look at a concurrent hash table implementation below.

```
#define BUCKETS (101)

typedef struct __hash_t {
    list_t lists[BUCKETS];
} hash_t;

void Hash_Init(hash_t *H) {
    int i;
    for (i = 0; i < BUCKETS; i++) {
        List_Init(&H->lists[i]);
    }
}

int Hash_Insert(hash_t *H, int key) {
    int bucket = key % BUCKETS;
    return List_Insert(&H->lists[bucket], key);
}

int Hash_Lookup(hash_t *H, int key) {
    int bucket = key % BUCKETS;
    return List_Lookup(&H->lists[bucket], key);
}
```

Figure 29.10: **A Concurrent Hash Table**

22 Conditional Variables

In many cases, a thread wants to make sure some **condition** is satisfied before continuing its execution. For example, think about *pthread_join()*, where a parent thread wants to wait for its **child** thread to complete. How do we implement such a routine? A simple, yet ineffective way of doing so is the good old **spin wait**, shown below.

```
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    return 0;
}
```

Figure 30.2: **Parent Waiting For Child: Spin-based Approach**

This methodology is grossly inefficient - as we've seen before, it can waste CPU cycles and can be incorrect in some scenarios.

22.1 Definition and Routine

In order to make a thread wait for a condition, we can use **condition variables**. A condition variable is simply an explicit queue that a thread can enter by calling a *wait()* function in order to wait on some condition. When another thread has satisfied this condition, it can *signal()* the condition variable and *wake up* the thread that was waiting in order to continue execution.

The *wait()* call (which is short for *pthread_cond_wait() in C*) accepts a reference to a lock as a parameter. This is because **the function actually frees the lock** as it waits for the condition to be satisfied. Thus,

it is assumed that the function will *only* be called when a thread *has the lock*. When another thread calls *signal()* (or *pthread_cond_signal()*), the *wait()* call **must reacquire the lock before returning**. Take a look at the implementation of a pthread *join()* function using condition variables.

```
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

Notice that we use a while loop when checking our flag. This is simply good practice and can prevent subtle race conditions when a thread must *wait()*.

Why do we need the *done* flag? Why can't we just use the *signal()* call by itself? What if we used the following code instead?

```
1
2  void thr\_exit() {
3      Pthread_mutex_lock(&m);
4      Pthread_cond_signal(&c);
5      Pthread_mutex_unlock(&m);
6      }
7
8  void thr\_join() {
9      Pthread_mutex_lock(&m);
10     Pthread_cond_wait(&c, &m);
11     Pthread_mutex_unlock(&m);
12 }
```

Listing 23: Neglecting the *done* flag

To see where this can cause problems, imagine the case where a a child thread executes before its parent. It will *signal()*, but no thread is waiting yet. When the parent thread reaches the conditional variable code, it will *wait()* and never be woken up.

Another thing to consider is *why we hold the lock for* these operations. What if we used the following lockless code?

```
1
2  void thr_exit() {
3      done = 1;
4      Pthread_cond_signal(&c);
5  }
6
7  void thr_join() {
8      if (done == 0)
9          Pthread_cond_wait(&c);
10 }
```

Listing 24: Neglecting the lock

If we just set a flag, we would incur possible **race conditions**. Imagine if the waiting thread was **interrupted** after seeing the *done* flag was 0, but *before* it called *wait()*. Then the signaling thread would signal before the waiting thread called *wait()*, so when the scheduler returns to the waiting thread, it just waits forever without ever being signaled!

22.2 The Producer/Consumer (Bounded Buffer) Problem

The next problem we will discuss is the **producer/consumer** problem, otherwise known as the **bounder buffer problem**. Imagine we have two sets of threads: **producers** that send data to a buffer, and **consumers** that grab and consume the data in some way. This scenario is relevant to real systems. Web servers accept and consume HTTP requests in this manner, and **pipes** are also a form of **bounded buffer**. In the following examples, we will be using *put()* and *get()* functions, which are implemented in the following way.

```
1   int buffer;
2   int count = 0; // initially, empty
3
4   void put(int value) {
5       assert(count == 0);
6       count = 1;
7       buffer = value;
8   }
9
10  int get() {
11      assert(count == 1);
12      count = 0;
13      return buffer;
14  }
```

Figure 30.4: **The Put And Get Routines (Version 1)**

With this implementation, we can only *put()* when the buffer is empty, and only *get()* when the buffer is full. Now let's take a look at a producer/consumer scenario using these functions below.

```
1   void *producer(void *arg) {
2       int i;
3       int loops = (int) arg;
4       for (i = 0; i < loops; i++) {
5           put(i);
6       }
7   }
8
9   void *consumer(void *arg) {
10      int i;
11      while (1) {
12          int tmp = get();
13          printf("%d\n", tmp);
14      }
15  }
```

Figure 30.5: **Producer/Consumer Threads (Version 1)**

Unfortunately, this code **does not work**. Even putting a lock around the *put()* and *get()* calls doesn't work, so we need **condition variables**. Below is an implementation doing so.

```
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 30.6: **Producer/Consumer: Single CV And If Statement**

Notice now that the consumer only grabs from the buffer when it is full, and the producer only writes to the buffer when it is empty. **Though this approach works well with one producer and one consumer, when there are multiple of either, there are two serious problems**.

The first of the two arises in the following scenario. Imagine we have two consumer threads and a single producer thread. Assume one of the consumer threads C_1 runs first due to the scheduler; it will find an empty buffer, start its *wait()* and defer the CPU to another thread. Eventually the producer will run and add to the buffer and *signal()* the condition variable showing the state has changed. Now when we we perform another context switch, we may switch to the second consumer who consumes the buffer and releases the lock. *Now, C_1, which already made it past the if condition, acquires the lock and attempts to consume the buffer as well, but its empty!*

This problem arises due to the fact that *the condition variable changed state again after being signaled*. Thus, there is no guarantee a state will not *change* before the corresponding thread wakes. This is known as **Mesa semantics**. **An easy fix is to always use while loop instead of an if statement when checking flags**. Now C_1 sees that the flag is set to only allow producer execution, and will wait again.

Another problem lies in the same scenario when the consumer signals the condition variable. Which thread, the producer or the consumer will acquire the lock? Obviously, the producer may be the chosen one, but this is up to the condition variable queue. *It may very well wake the other consumer thread, which would see that the flag is not set and sleep.* **Now all three threads are asleep and nothing will wake them up! Yikes!**

Fortunately, there is yet another simple fix here. **We can use two different condition variables, one for signaling the producers and one for signaling the consumers.** This way, when a consumer thread signals the conditional variable, it will definitely wake up a producer and the threads cannot all concurrently wait. Finally, we can implement a correct producer/consumer system, shown below.

Notice that we have changed the *put()* and *get()* functions to be able to write/read multiple times to the buffer. *This is much better for eliminating context switching overhead and making better use of our CPU cycles.* Also, now threads can read and write to the buffer *at the same time*, greatly increasing our **concurrency**.

```
int buffer[MAX];
int fill_ptr = 0;
int use_ptr  = 0;
int count    = 0;

void put(int value) {
    buffer[fill_ptr] = value;
    fill_ptr = (fill_ptr + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use_ptr];
    use_ptr = (use_ptr + 1) % MAX;
    count--;
    return tmp;
}
```

Figure 30.11: **The Correct Put And Get Routines**

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);             // p1
        while (count == MAX)                    // p2
            Pthread_cond_wait(&empty, &mutex); // p3
        put(i);                                 // p4
        Pthread_cond_signal(&fill);             // p5
        Pthread_mutex_unlock(&mutex);           // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);             // c1
        while (count == 0)                      // c2
            Pthread_cond_wait(&fill, &mutex);  // c3
        int tmp = get();                        // c4
        Pthread_cond_signal(&empty);            // c5
        Pthread_mutex_unlock(&mutex);           // c6
        printf("%d\n", tmp);
    }
}
```

Figure 30.12: **The Correct Producer/Consumer Synchronization**

22.3 Covering Conditions

Let's take a look at the following example of using **condition variables** in a memory allocation library.

```
// how many bytes of the heap are free?
int bytesLeft = MAX_HEAP_SIZE;

// need lock and condition too
cond_t  c;
mutex_t m;

void *
allocate(int size) {
    Pthread_mutex_lock(&m);
    while (bytesLeft < size)
        Pthread_cond_wait(&c, &m);
    void *ptr = ...; // get mem from heap
    bytesLeft -= size;
    Pthread_mutex_unlock(&m);
    return ptr;
}

void free(void *ptr, int size) {
    Pthread_mutex_lock(&m);
    bytesLeft += size;
    Pthread_cond_signal(&c); // whom to signal??
    Pthread_mutex_unlock(&m);
}
```

Figure 30.13: **Covering Conditions: An Example**

Let's consider the scenario where there are zero available bytes of memory left. Let's say the first thread T_1 attempts to allocate 100 bytes, and a second thread tries to allocate only 10. Both will *wait()* due to the lack of available memory.

However, what happens when another thread calls *free()* and frees **50** bytes of memory. Only T_1 should wake up, but it might choose to wake up T_2 instead. Thus, **the signaling thread doesn't know which thread to wake up**.

This scenario was fixed, though not efficiently, by allowing for a condition variable ***broadcast()***, that **wakes all threads**. Now T_1 is guaranteed to be woken up, but we've greatly hindered our performance by waking *every* thread. This scenario is called a **covering condition**, as it covers all the cases where a thread needs to wake up.

23 Semaphores

In many cases, we need both **locks** and **condition variables** in order to implement some concurrent functionality. Thus, a combination of both called **semaphores**, were created.

23.1 Semaphores: A Definition

A semaphore is an object with an integer value that we can manipulate with two routines: *sem.wait()* and *sem.post()*. The general ideas for these methods are shown below.

```
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**

In addition, we must initialize a semaphore with a starting value, where we call *sem.init()*. We pass this value to the third parameter, a reference to the semaphore variable to the first value, and the second value is usually set to 0 (unless we are synchronizing amongst multiple processors).

23.2 Binary Semaphores

A **binary semaphore** is simply a semaphore that is used to implement a lock. It is dubbed "binary" because locks can only have two values: held and not held. Such an implementation is shown below.

```
1 {
2     sem_t m;
3     sem_init(&m, 0, 1);
4
5     sem_wait(&m);
6     // critical section here
7     sem_post(&m);
8 }
```

Listing 25: A binary semaphore (lock)

Below is a sample thread trace of how this would perform using a single thread.

Value of Semaphore	Thread 0	Thread 1
1		
1	call `sem_wait()`	
0	`sem_wait()` returns	
0	`(crit sect)`	
0	call `sem_post()`	
1	`sem_post()` returns	

Figure 31.4: **Thread Trace: Single Thread Using A Semaphore**

Here is trace of the same implementation, but using two threads.

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call `sem_wait()`	Running		Ready
0	`sem_wait()` returns	Running		Ready
0	`(crit sect: begin)`	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	call `sem_wait()`	Running
-1		Ready	`decrement sem`	Running
-1		Ready	`(sem<0)→sleep`	Sleeping
-1		Running	*Switch→T0*	Sleeping
-1	`(crit sect: end)`	Running		Sleeping
-1	call `sem_post()`	Running		Sleeping
0	`increment sem`	Running		Sleeping
0	`wake(T1)`	Running		Ready
0	`sem_post()` returns	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	`sem_wait()` returns	Running
0		Ready	`(crit sect)`	Running
0		Ready	call `sem_post()`	Running
1		Ready	`sem_post()` returns	Running

Figure 31.5: **Thread Trace: Two Threads Using A Semaphore**

23.3 Semaphores For Ordering

We can also use semaphores for **ordering**, almost identically to how **condition variables** work. An implementation of this is shown below.

```
1  sem_t s;
2
3  void *
4  child(void *arg) {
5      printf("child\n");
6      sem_post(&s); // signal here: child is done
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     sem_init(&s, 0, X); // what should X be?
13     printf("parent: begin\n");
14     pthread_t c;
15     Pthread_create(&c, NULL, child, NULL);
16     sem_wait(&s); // wait here for child
17     printf("parent: end\n");
18     return 0;
19 }
```

Figure 31.6: **A Parent Waiting For Its Child**

When using a semaphore for ordering in this case, **its initial value should be set to 0, not 1**. Because the semaphore starts off at zero, the parent either decrements the semaphore first, then waiting for the child process to increment it and signal it to wake up **OR** the child process runs first and calls *sem.post()*, incrementing the semaphore so when the parent calls *sem.wait()* and decrements the semaphore, the value is still non-negative and it can return immediately.

Thus, the parent thread waits for the child to complete. In order to see why, take a look at the two possible thread traces of the above program.

Value	Parent	State	Child	State
0	`create(Child)`	Running	*(Child exists; is runnable)*	Ready
0	call `sem_wait()`	Running		Ready
-1	`decrement sem`	Running		Ready
-1	`(sem<0)`→`sleep`	Sleeping		Ready
-1	*Switch→Child*	Sleeping	`child runs`	Running
-1		Sleeping	call `sem_post()`	Running
0		Sleeping	`increment sem`	Running
0		Ready	`wake(Parent)`	Running
0		Ready	`sem_post()` returns	Running
0		Ready	*Interrupt; Switch→Parent*	Ready
0	`sem_wait()` returns	Running		Ready

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

Value	Parent	State	Child	State
0	`create(Child)`	Running	*(Child exists; is runnable)*	Ready
0	*Interrupt; Switch→Child*	Ready	`child runs`	Running
0		Ready	call `sem_post()`	Running
1		Ready	`increment sem`	Running
1		Ready	`wake(nobody)`	Running
1		Ready	`sem_post()` returns	Running
1	`parent runs`	Running	*Interrupt; Switch→Parent*	Ready
1	call `sem_wait()`	Running		Ready
0	`decrement sem`	Running		Ready
0	`(sem≥0)`→`awake`	Running		Ready
0	`sem_wait()` returns	Running		Ready

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

23.4 The Producer/Consumer (Bounded Buffer) Problem

Another problem we can solve with semaphores is the **producer/consumer** problem. Take a look at the following implementations of the consumer and producer functions shown below (the *get()* and *put()* functions are the same as before).

```
1  sem_t empty;
2  sem_t full;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          sem_wait(&empty);           // Line P1
8          put(i);                     // Line P2
9          sem_post(&full);            // Line P3
10     }
11 }
12
13 void *consumer(void *arg) {
14     int i, tmp = 0;
15     while (tmp != -1) {
16         sem_wait(&full);            // Line C1
17         tmp = get();                // Line C2
18         sem_post(&empty);           // Line C3
19         printf("%d\n", tmp);
20     }
21 }
22
23 int main(int argc, char *argv[]) {
24     // ...
25     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
26     sem_init(&full, 0, 0);    // ... and 0 are full
27     // ...
28 }
```

Figure 31.10: **Adding The Full And Empty Conditions**

This code works well problem with this code is that there is a race condition, specifically in the *put()* and *get()* operations. Two producers could enter the *put()* function and exhibit such a race condition, causing the buffer to be overwritten. Thus, we need to implement a **binary semaphore** in order to ensure **mutual exclusion** in this **critical section**.

A correct implementation is shown below. **Note that the binary semaphore is *inside* of the ordering semaphore**. If this was not the case, our code falls prey to **deadlock**, where both the producer and consumer will *wait()* forever.

```
1   sem_t empty;
2   sem_t full;
3   sem_t mutex;
4
5   void *producer(void *arg) {
6       int i;
7       for (i = 0; i < loops; i++) {
8           sem_wait(&empty);           // Line P1
9           sem_wait(&mutex);           // Line P1.5 (MOVED MUTEX HERE...)
10          put(i);                     // Line P2
11          sem_post(&mutex);           // Line P2.5 (... AND HERE)
12          sem_post(&full);            // Line P3
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          sem_wait(&full);            // Line C1
20          sem_wait(&mutex);           // Line C1.5 (MOVED MUTEX HERE...)
21          int tmp = get();            // Line C2
22          sem_post(&mutex);           // Line C2.5 (... AND HERE)
23          sem_post(&empty);           // Line C3
24          printf("%d\n", tmp);
25      }
26  }
27
28  int main(int argc, char *argv[]) {
29      // ...
30      sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31      sem_init(&full, 0, 0);    // ... and 0 are full
32      sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
33      // ...
34  }
```

Figure 31.12: **Adding Mutual Exclusion (Correctly)**

23.5 Reader-Writer Locks

Sometimes we want to implement locks in a way that takes more advantage of concurrency. For example, if we have a list data structure, we can have as many threads *reading* the list as we want **as long as no thread is writing to the list** (race conditions!). Thus, take a look at the following **reader-writer lock**.

```
1  typedef struct _rwlock_t {
2    sem_t lock;      // binary semaphore (basic lock)
3    sem_t writelock; // used to allow ONE writer or MANY readers
4    int   readers;   // count of readers reading in critical section
5  } rwlock_t;
6
7  void rwlock_init(rwlock_t *rw) {
8    rw->readers = 0;
9    sem_init(&rw->lock, 0, 1);
10   sem_init(&rw->writelock, 0, 1);
11 }
12
13 void rwlock_acquire_readlock(rwlock_t *rw) {
14   sem_wait(&rw->lock);
15   rw->readers++;
16   if (rw->readers == 1)
17     sem_wait(&rw->writelock); // first reader acquires writelock
18   sem_post(&rw->lock);
19 }
20
21 void rwlock_release_readlock(rwlock_t *rw) {
22   sem_wait(&rw->lock);
23   rw->readers--;
24   if (rw->readers == 0)
25     sem_post(&rw->writelock); // last reader releases writelock
26   sem_post(&rw->lock);
27 }
28
29 void rwlock_acquire_writelock(rwlock_t *rw) {
30   sem_wait(&rw->writelock);
31 }
32
33 void rwlock_release_writelock(rwlock_t *rw) {
34   sem_post(&rw->writelock);
35 }
```

Figure 31.13: **A Simple Reader-Writer Lock**

In this implementation, we use a *write_lock* that can only be held by either the writers or the readers. Thus, **no two threads can read and write at the same time**. When the number of readers hits 1, the readers attempt to acquire the write lock, and they keep it until there are no threads reading (where readers = 0).

Note: Reader-writer locks can be very **unfair**. The readers can easily monopolize the lock, and thus the writers will never obtain a chance to do their job. In addition, reader-writer locks can incur lots of **overhead**, causing them to not be as useful as one would think.

23.6 How to Build Semaphores

Shown below is a possible implementation of a **semaphore** using our previous synchronization primitives, **locks** and **condition variables**.

```
1  typedef struct __Zem_t {
2      int value;
3      pthread_cond_t cond;
4      pthread_mutex_t lock;
5  } Zem_t;
6
7  // only one thread can call this
8  void Zem_init(Zem_t *s, int value) {
9      s->value = value;
10     Cond_init(&s->cond);
11     Mutex_init(&s->lock);
12 }
13
14 void Zem_wait(Zem_t *s) {
15     Mutex_lock(&s->lock);
16     while (s->value <= 0)
17         Cond_wait(&s->cond, &s->lock);
18     s->value--;
19     Mutex_unlock(&s->lock);
20 }
21
22 void Zem_post(Zem_t *s) {
23     Mutex_lock(&s->lock);
24     s->value++;
25     Cond_signal(&s->cond);
26     Mutex_unlock(&s->lock);
27 }
```

Figure 31.16: **Implementing Zemaphores With Locks And CVs**

24 Common Concurrency Problems

24.1 Types of Concurrency Bugs

We've already discusses many concurrency problems in the previous chapters, one being the infamous **deadlock**. We can actually group our concurrency problems into two groups **deadlock bugs** and **non-deadlock bugs**.

24.2 Non-deadlock bugs

There are two types of **non-deadlock bugs**: **atomicity violation** and **order violation**. Let's take a look at both.

24.2.1 Atomicity Violation

Atomicity violation is essentially when a code segment is meant to be **atomic** - that is, executed as if it were a single instruction, but this is not the case. It can lead to nasty concurrency issues, as shown in the scenario below.

```
1 {
2 Thread 1::
3     if (thd->proc_info) {
4         ...
5         fputs(thd->proc_info, ...);
6         ...
7     }
```

```
8
9  Thread 2::
10     thd->proc_info = NULL;
11 }
```

Listing 26: An example of atomicity violation

In this code, if the first thread were to enter the conditional statement and then become interrupted, the second thread could set *proc_info* pointer to null and cause problems for the first thread. This could be easily fixed with a **lock** or **mutex**. This is an example of an **atomicity violation**.

24.2.2 Order Violation

Order violation is a pretty intuitive concept as well. Essentially, it's when some control flow *must* be executed in a particular order, but concurrency causes a misordering of such an execution. An example is shown below.

```
1  {
2  Thread 1::
3      void init() {
4      ...
5      mThread = PR_CreateThread(mMain, ...);
6      ...
7      }
8
9  Thread 2::
10     void mMain(...) {
11     ...
12     mState = mThread->State;
13     ...
14     }
15 }
```

Listing 27: An example of order violation

In this example, thread 2 obviously needs to wait for the initialization step before it accesses the *mThread* structure. It is very possible that thread 2 runs before thread 1, and so this order may be violated. The fix is simple, however: just throw in a **condition variable**!

24.3 Deadlock bugs

24.3.1 What is a deadlock?

Deadlock bugs are a bit more convoluted than their counterparts. A **deadlock** occurs when multiple threads or holding locks that the other is waiting for, and so neither will obtain the lock and they will both be caught at a standstill! Below is an example of a scenario where this could happen.

```
1  {
2  Thread 1::
3      pthread_mutex_lock(L1);
4      pthread_mutex_lock(L2);
5
6  Thread 2::
7      pthread_mutex_lock(L2);
8      pthread_mutex_lock(L1);
9  }
```

Listing 28: An example of order violation

If thread 1 gets preempted before acquiring the second lock, thread 2 could grab it and begin waiting for the L1 lock. Because thread 1 is waiting for the L2 lock, it will never give up the L1 lock, and so both threads are stuck! The diagram below shows this cyclical dependency.

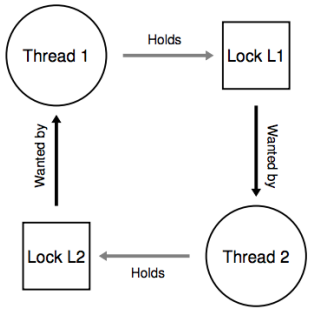


Figure 32.2: **The Deadlock Dependency Graph**

24.3.2 Why do deadlocks occur?

The situation above seems like it could be easily corrected by having both threads acquire the locks in the same order, but in real life things aren't so simple. There are multiple reasons that deadlocks can occur without foolish implementations.

- *Complex Dependencies*: in large-scale code bases, underlying complex dependencies can exist between components. For example, an OS's virtual memory system may need to access the file system to page in a block of memory, but when paging that block in, the file system may need to contact virtual memory when reading the block into another page.
- *Encapsulation*: software developers often hide details of implementations in order to bolster abstraction. However, underlying dependencies within these implementations are not always clear within the interface specifications, and so deadlocks can occur when using such implementations.

24.3.3 Conditions for Deadlock

Four conditions are necessary for *any deadlock* to occur.

- **Mutual exclusion:** Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).
- **Hold-and-wait:** Threads hold resources allocated to them (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire).
- **No preemption:** Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.
- **Circular wait:** There exists a circular chain of threads such that each thread holds one or more resources (e.g., locks) that are being requested by the next thread in the chain.

If *any* of these conditions is not satisfied, a deadlock cannot occur. Thus, preventive deadlock solutions are aimed at eliminating one of these conditions.

24.3.4 Circular Wait Prevention

One of the most practical prevention techniques of deadlocking is to structure locking code so that there is never a **circular wait** - that is, always acquire the locks in some order. This is called **total ordering**.

In more complex systems, there are going to be many locks and **total ordering** isn't always a feasible option. In this case, **partial ordering** may be more beneficial. For example, acquiring lock *L1* before *L2* is a partial ordering, and so is *L1* before *L2* before *H1* before *H2*.

24.3.5 Hold-and-wait Prevention

The **hold-and-wait** requirement for deadlocks can be easily solved by acquiring all locks atomically, perhaps with another lock, as shown below.

```
{
    pthread_mutex_lock(prevention); // begin lock acquisition
    pthread_mutex_lock(L1);
    pthread_mutex_lock(L2);
    ...
    pthread_mutex_unlock(prevention); // end
}
```

Listing 29: Hold-and-wait prevention

Unfortunately, this approach has its flaws. It requires us to acquire all locks ahead of time and know which locks must be held, which **encapsulation** works against. Also, acquiring all the locks at once limits concurrency quite a bit.

24.3.6 No Preemption Prevention

Because we assume that a lock is held until *unlock* is called, multiple lock acquisition can be problematic when holding one lock and attempting to acquire another. Many thread libraries provide flexible lock acquisition methods to combat this and add some degree of preemption. Take a look below at such an implementation.

```
{
top:
    pthread_mutex_lock(L1);
    if (pthread_mutex_trylock(L2) != 0) {
        pthread_mutex_unlock(L1);
        goto top;
    }
}
```

Listing 30: No preemption prevention

The *pthread_mutex_trylock()* function attempts to acquire a lock, and if it cannot, then the thread releases its held lock and tries again, perhaps after a preemption.

One downfall of this implementation is the introduction of a new problem: **livelock**. It is possible, though improbable, that two threads continuously attempt and fail to acquire both locks. No progress is being made here even though it is not technically a deadlock.

Another downfall of this implementation is once again **encapsulation**. This scenario only has 2 locks, but in complex methods, we need to have knowledge of all lock acquisitions and be careful to release all of them before jumping back to the top. In addition, if any memory was allocated, it would need to be released as well. However, in limited scenarios it could be a good fix.

24.3.7 Mutual Exclusion Prevention

Another way to prevent deadlocking is to prevent any need for mutual exclusion at all. However, this is difficult, as we know we have **critical sections**. But it is possible through the use of hardware. In fact, it is possible to create **lock-free/wait-free** data structure implementations using such hardware instructions. Recall a linked list insert method involving locks to avoid **race conditions**, shown below.

```
void insert(int value) {
    node_t *n = malloc(sizeof(node_t));
    assert(n != NULL);
    n->value = value;
    pthread_mutex_lock(listlock); // begin critical section
    n->next = head;
    head = n;
    pthread_mutex_unlock(listlock); // end critical section
```

```
}
```

Listing 31: Thread-safe linked list insert method w/ mutex

Now take a look at a lockless, yet still thread-safe implementation using the compare and swap hardware instruction.

```
void insert(int value) {
    node_t *n = malloc(sizeof(node_t));
    assert(n != NULL);
    n->value = value;
    do {
    n->next = head;
    } while (CompareAndSwap(&head, n->next, n) == 0);
}
```

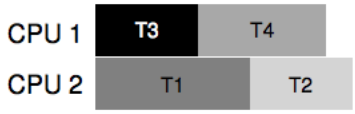
Listing 32: Thread-safe linked list insert method w/ mutex

Implementing delete, find, and insert methods for the linked list data structure is non-trivial and can be complicated, and so implementing lock-free data structures is a strenuous task in itself.

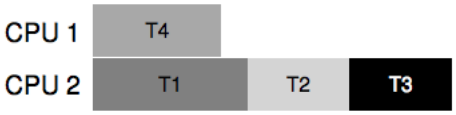
24.3.8 Deadlock Avoidance via Scheduling

Another option in combating deadlocks is deadlock **avoidance**, where we don't allow strategically schedule threads based on global knowledge of which locks each thread will acquire.

For example, say we have four threads running on two CPUs, and throughout their executions thread T1 acquires locks *L1* and *L2*, thread *T2* also acquires *L1* and *L2*, thread *T3* only acquires *L1* and *T4* doesn't acquire any locks. If threads *T1* and *T2* never run concurrently, there is *no chance of a deadlock*. Smart scheduling of the threads would avoid any deadlocks. Such a scheduling is shown below.



First of all, this approach is only viable in limited cases where we have global knowledge of lock acquisition. Second of all, this approach can greatly limit performance. Take a look at the following scenario, where thread *T1* also now grabs locks *L1* and *L2*.



As you can see, this static scheduling can cause performance issues. As such, this approach to avoiding deadlocks also has its drawbacks.

24.3.9 Detect and Recover

One final way of dealing with deadlocks is to just let them happen. If deadlocks occur very infrequently, simply restarting the system can be pragmatic. For this approach, there needs to be some sort of deadlock detector, that holds a resource graph and looks for cycles. Though it may seem lazy, this approach can be simple and sweet for many cases.