Operating Systems: Three Easy Steps

Kunwar Shaanjeet Singh Grover

Contents

Chapter 1

Introduction

1.1 Virtualisation

The OS takes a **physical** resources (such as the processor , or memory, or a disk) and transforms it into a more general, powerful, and easy-to-use **virtual** for of itself. Thus, we sometimes refer to the operating system as a **virtual machine**. This general technique of transforming is called **virtualisation**.

1.1.1 CPU Virtualisation

Turning a single CPU into a seemingly infinite number of CPUs and thus allowing many programs to seemingly run at once is what we call **virtualizing the CPU**.

1.1.2 Memory Virtualisation

Each process accesses its own **virtual address spaces**, which the OS somehow maps onto the physical memory of the machine. Exactly how this is accomplished is what we study.

1.2 Concurrency

Concurrency is a conceptual term to refer to a host of problems that arise, and must bf addressed when working on many things at once (i.e. concurrently) in the same program.

1.3 Presistence

The software in the operating system that usually manages the disk is called the **file system**; it is this responsible for storing any files the user creates in a reliable and efficient manner on the disks of the system. The file system is the part of OS in charge of managin persisten data. What techniques are needed to do so? What mechanisms and policies are required to do so with a high probability?

Chapter 2

CPU Virtualisation

2.1 The Abstraction: The Process

Process: A running program.

Crux Of The Problem: Although there are only a few physical CPUs available, how can the OS provide the ullusion of nearly-endless supply of said CPUs?

The OS creates this illusion by **virtualizing** the CPU. By running one process, then stopping it and running another, and so forth, the OS can promote the illusion that many virtual CPUs exist when in fact there is only one physical CPU (or a few). This technique is called **time-sharing**.

To do this, OS requires some low-level machinery and some high-level intelligence. The low-level machinery is called **mechanisms**. For example, stopping one program and starting another on a given CPU. On top of these mechanisms, there is high-level intelligence call **policies**. Policies are algorithms for making decisions within the OS.

Mechanisms: Provides answer to a *how* question.
Policies: Provides answer to a *which* question.

2.2 Mechanism: Limited Direct Execution

The Crux: How to efficiently virtualize the CPU with control?

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

2.2.1 Basic Technique: Limited Direct Execution

The *direct execution* part of the idea: just run the program directly on the CPU. Thus, when the OS wishes to start a program, it creates a process entry for it in a process list, allocates some memory for it, loads the program code into memory, locates its entery point (main()), jumps to it, and starts running the user's code.

This approach gives rise to a few problems:

- How to make sure that the program does not do anything that we dont want it to do while still maintaining efficency?
- How do we stop a process and switch to another process, i.e. how to implement **time sharing** we require to virtualize the CPU?

Problem #1: Restricted Operations

What if the process wishes to perform some kind of restricted operation, such as issuing an I/O request to a disk, or gaining access to more system resources such as CPU or memory?

The Curx: How to perform restricted operations
A process must be able to perform I/O and some other restricted operations, but without giving the process complete control over the system.

Two differend modes:

- **User Mode:** Code that runs in user mode is restricted in what it can do. For example, when running in user mode, a process can't issure I/O requests; doing so would result in the processor raising an exception.
- **Kernal Mode:** Mode the operating system (or kernal) runs in. In this mode, code that runs can do whatever it likes, including privileged operations such as issuing I/O requests and executing restricted instructions.

All modern hardware provides the ability for user programs to perform a **system call**. System calls allow the kernal to carfully expose certain key pieces of functionality to user programs such as accessing the file system, creating and destroying the processes, communicationg with other processes, and allocating more memory.

System calls To execute a system call, a program must execute a special **trap** instruction. This instruction jumps into the kernal and raises privilege level to kernal mode; once in the kernal, the system can now perform the restricted operations needed and thus the required work for the calling process. When finished, the OS calls a special **return-from-trap** instruction, which returns into the calling user program while simultaneously reducing the privilege level back to user mode.

When executing a trap, the caller's registers must be saved in order to return correctly when the OS issues the return-from-trap instruction.

On x86, the processor will push the program counter, flags and a few other registers onto a per-process **kernal stack**; return-from-trap will pop these values from the stack and resume execution.

How does the trap know which code to run inside the OS? Clearly, the calling process can't specify and address to jump to; doing so would allow programs to jump anywhere into the kernal which is not secure. The kernal must control what code executes upon a trap.

The kernal does so by setting up a **trap table** at boot time. When the machine boots up, it does so in kernal mode. One of the first things the OS does is to tell the hardware what code to run when certain exceptional events occur. The OS informs the hardware of these locations of these **trap handlers**, usually with some kind of special instruction. Once the hardware is informed, it remembers the location of these handlers until the machine is rebooted, and thus hardware knows what to do when system calls and other exceptional events take place.

To perform the exact system call, a **system-call number** is usually asigned to each system call. The user code has to place the desired system-call number in a register or at a specific location in the stack; the OS, when handeling the system call inside the trap handler, examines the number, ensures it is valid, and, if it is, executes the corresponding code. This level of indirection serves as a form of **protection**; user code cannot specify an exact address to jump to, but rather must request a particular service via a number.

Telling the hardware where the trap tables are is a **privileged** operationg, otherwise you could make your own trap tables and compromise the system.

How all this works:

1. At boot time, the kernal initializes the trap table, and the CPU remembers its location for subsequent use. The kernal does so via a privileged instruction.

2. The kernal sets up a few things (allocating memory, etc.) before using a return-from-trap instruction to start the execution of the process. When the process wishes to issue a system call, it traps back into the OS, which handles it and once again returns control via a return-from-trap to the process. The process completes its work and returns from main().

Problem #2: Switching Between Processes

When a process is running on the CPU, this by definition means the OS is *not* running. If OS isnt running how can it change a process? (it cant).

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

Cooperative Approach: Wait for system calls In this approach, the OS regains control of the CPU by waiting for a system call or an illegal operation of some kind to take place, as control is passed to the OS during these exceptions.

Non-Cooperative Approach: The OS takes control If we get stuck in an infinite loop in the cooperative approach, the only way out is to reboot the machine.

The Crux: How to gain control without cooperation
How can the OS gain control of the CPU even if processes are not being cooperative? What can the OS do to ensure a rogue process does not take over the machine?

Timer interrupt: A timer device is programmed to raise an interrupt after a fixed delay; when the interrupt is raised, the currently running process is halted, and a pre-configured **interrupt handler** in the OS runs. Now the OS has regained control of the CPU, and thus can stop the current process and start a different one.

At boot time, the OS starts the interrupt timer. This is a privileged operation.

Saving and Restoring Context The OS has regained control now. It can decide to switch or not. This decision is made by the **scheduler**.

If the decision is to switch, the OS executes a **context switch**. A context switch saves register values of the currently-executing process and restores register values for the soon-to-be-executing process. After this, the OS executes a return-from-trap instruction and executes the new process.

By switching stacks pointers (stack pointer is a register), the kernel enters call to the switch code in the context of one process (the interrupted one) amd returns in the context of another (the soon-to-be-executing one). By switching stack is actually how the process is switched.

There are two types of register saves/restores that happen here:

- When the timer interrupt occurs. The *user registers* of the running process are implicitly saved by the *hardware*, using the kernal stack of the process.
- When OS performs the context switch. The *kernal registers* are explicitly saved by the *software* (i.e. th OS), but this time into memory in the process structure of the process.

More info on the last two points here

2.3 Scheduling

In this section we will understand **policies** that OS scheduling employs.

The Crux: How To Develop Scheduling Policy

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

We will make some assumptions and continue to relax them to get to an optimal policy. Assumptions:

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

For now, we will use a single metric to determine optimal policy: **turnaround time**.

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

2.3.1 First In, First Out (FIFO)

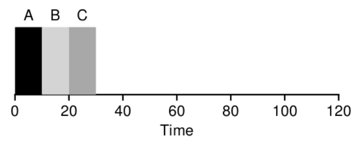


Figure 2.1: FIFO Simple Example

For our current assumptions, this is optimal. But if we relax the assumption that job running time are same, we can get bad turnaround time:

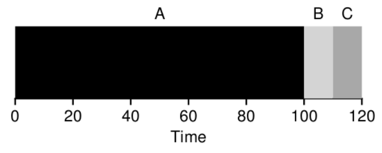


Figure 2.2: Why FIFO is Not that Great

2.3.2 Shortest Job First (SJF)

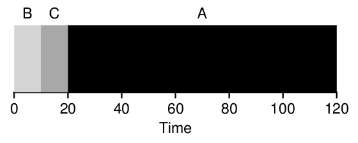


Figure 2.3: SJF Simple Example

It can be proven that under these assumptions, SJF will give the best turnaround time.

Now, lets relax the assumption that all jobs arrive at the same time. Now, we can build a worst case for SJF:

2.3.3 Shortest Time-To-Completion First (STCF)

To address this concern, we need to relax assumption 3 (jobs started, must run to completion). SJF by our defination was a **non-preemptive** scheduler, and thus suffered from the problems in figure 2.4.

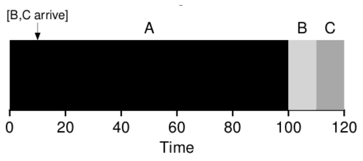


Figure 2.4: SJF With Late Arrivals from B and C

To improve upon that, we add preemption to SJF, known as STCF. Any time a new job enters the system, the STCF schedular determines which of the remaining jobs has least time left, and schedules that. It can be proven that under the current assumptions, this is the best schedular (w.r.t turnaround time)

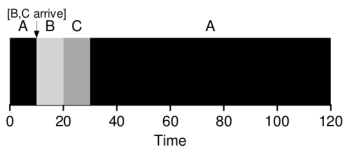


Figure 2.5: STCF Simple Example

New Metric: Response Time: If our only metric was turnaround time and we knew job runtime, STCF would be a great policy. But inroduction of time-sharing machines, required interactive performance from sustem as well. Thus a new metric was needed: **response time**

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

2.3.4 Round Robin

To improve upon response time, we introduce **Round-Robin (RR)** scheduling. Basic idea: Instead of running a job to completion, RR runs a job for a **time slice** (or **scheduling quantum**) and then switches to the next job in the run queue.

The length of time-slice is obviously a multiple of timer-interupt.

The length of time slice is critical for RR. The shorter it is, the better performance of RR under response-time metric. However, too small time-slice can result in time taken to context switch to dominate overall performance. Thus, time slices hould be made long enough to **amortize** the cost of switching without making it so long that system is no longer responsive.

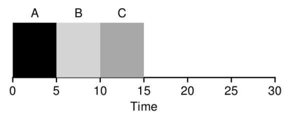


Figure 2.6: SJF (Bad response time)

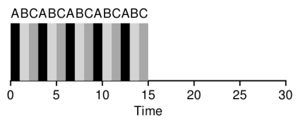


Figure 2.7: RR (Good response time)

RR is one of the *worst* policies for turnaround time because it extends the program to as long as possible.

Incorporating I/O

We now relax assumption that there is no I/O. When a job initiates an I/O request, because the currently-running job won't be using the CPU during the I/O; it is **blocked** waiting for the I/O completion. The CPU is idle during the time of the I/O.

The scheduler incorporates I/O by treating each CPU burst as a job. the scheduler makes sure processes that are ”interactive” get run frequently. While those interactive jobs are performing I/O, other CPU-intensive jobs run, thus better utilizing the processor.

2.3.5 The Multi-Level Feedback Queue

In this subsection, we remove the assumption that we know job time and we will describe the most well-known approach to scheduling, **MLFQ**.

The Crus: How To Schedule Without Perfect Knowledge?

How can we design a schedular that both minimizes response time for interactive jobs while also minimizing turnaround time without a *prior* knowledge of job length?

MLFQ has a number of distinct **queues**, each assigned a different **priority level**. MLFQ uses priorities to decide which job should run at a given time.

Basic rules for MLFQ:

1. If Priority(A) > Priority(B), A runs (B doesn't)

2. If Prority(A) = Priority(B), A & B run in RR.

Rather than giving a fixed priority to each job, MLFQ *varies* the priority of a job based on its *observed behavior*.

If a job repeatedly relinquishes the CPU while waiting for input from the keyboard, MLFQ will keep its priority high, as this is how an interactive process might behave. If a job uses the CPU intensively for long periods of time, MLFQ will reduce its priority. In this way, MLFQ will try to *learn* about processes as they run, and thus use the *history* of the job to predict its *future* behavior.

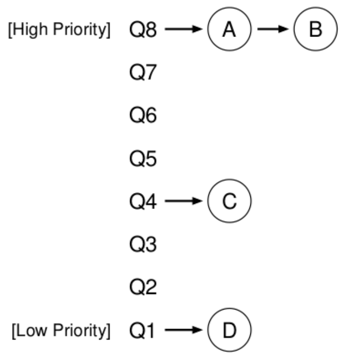


Figure 2.8: MLFQ Example

Attempt #1: How to Change Priority

Keep in mind: Our workload contains a mix of interactive jobs taht are short running (and may frequently relinquish the CPU), and some longer running "CPU-bound" jobs that need a lot of CPU time but where the response time isn't important. Here is the first attempt at an algoritm:

3. When a job enters the system, it is placed at highest priority (the topmost queue)

4.
    - If a job uses up an entire time slice while running, its priority is *reduced*.
    - If a job gives up the CPU before the time slice is up, it staus at the *same* priority level.

2.3.6 Problems with current MLFQ

There is a problem of **starvation:** if there are ”too many” interactive jobs in the system, they will combine to consume *all* CPU time, and thus long-running jobs will *never* receive any CPU time.

Another problem is that the schedular can be **gamed**. Gaming a schedular refers to the idea to tricking the schedular in giving more time than the job's share. Before the time-slice is over, the job could issue and I/O and remain at the same priority. Doing so, allows the program to gain a higher percentage of CPU time.

2.3.7 Attemp #2: The Priority Boost

We add a new rule to handle the flaws.

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

This solves the problem of starvation. S has to be set properly.

2.3.8 Attempt #3: Better Accounting

We rewrite Rule 4 to make it anti-gaming.

4. Once a job uses its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced.

MLFQ: Summary

1. If Priority(A) > Priority(B), A runs (B doesn't)
2. If Priority(A) = Priority(b), A & B run in round-robin fashion using the time slice of the given queue.
3. When a job enters the system, it is placed at the highest priority.
4. Once a job uses up its time allotment at a given level, its priority is reduced.
5. After some period S, move all jobs to the topmost queue.

Chapter 3

Memory Virtualization

3.1 The Abstraction: Address Space

Address Space: The running program's view of memory in the system

The Crux: How to Virtualize Memory?

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

Goals of OS while virtualizing memory:

1. Transparency: How OS implements virtual memory should be invisible to the running program.
2. Efficiency
3. Protection: A process should not be able to access memory out of its virtual memory

3.2 Mechanisms

The Crux: How to Efficiently And Flexibly Virtualize Memory

How can we build an efficent virtualization of memory? How do we maintain control over the locations application can access? how do we do all of this efficently?

How it will work:

- Hardware provides mechanisms for **hardware-based address translation**. Hardware only provides the low-level mechanisms.
- OS will use these mechanisms to manage memory, keeping track of free memory and maintain control.

Assumptions we make for now:

- Address space must be placed *contiguously* in physical memory.
- Size of address space is less than size of physical memory.

We will continue to relax these assumptions as we develop a better model. The current model will be laughable at best.

What the process should see:

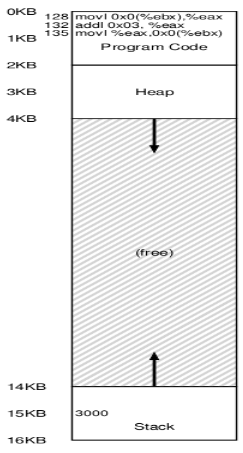


Figure 3.1: A process and its address space

What should actually go in memory:
The program thinks its address space starts at address 0, while it actually starts at 32KB in the physical memory.

3.2.1 Dynamic(Hardware-based) Relocation

We discuss the furst incarnation of address translation: **base and bounds**.

Specifically, we'll need two hardware registers within each CPU: one called the **base** register, and the other the **bounds** register. Using base and bounds, we can place the address space anywhere we like in the physical memory and ensure that it only has access to its own address space.

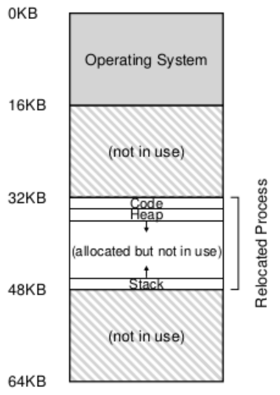


Figure 3.2: Physical Memory with the process

In this setup, each program is written and compiled as if it is loaded at address zero. However, when running the OS decides where in physical memory to place the program and sets the base register to that value. Now, the physical address can be calculated as:

$$physical\ address = virtual\ address + base$$

Memory refrences generated by the process are **virtual addresses**. The hardware turns these virtual addresses into **physical addresses**. This technique of transforming a virtual address is exactly what is refered as **address translation** The bounds register is used to check if the virtual address lies in the virtual memory or not.

Since this relocation happens at runtime and we can move address spaces after process has started runnning, this technique is reffered to as **dynamic relocation**

Note that base and bounds registers are hardware structures kep on the CPU chip. This part of processor which helps with address translation is called **Memory Management Unit (MMU)**. We will continue to add more circuity to MMU.

Hardware Support Summary till now

- **Privileged mode**: Needed to prevent user-mode processes from executing privileged instructions
- **Base/bounds registers**: pair of registers per CPU to support address translation and bound checks

- **Privileged instructions to update base/bounds**
- **Privileged instructions to register exception handlers**: How to handle exceptions must be told by the OS
- **Ability to raise exceptions**

OS Requirement Summary for memory

- **Memory Management**: Need to allocate memory for new processes, reclaim memory from terminated processes, manage memory via **free list**.
- **Base/bounds management**: Must set base/bounds properly upon context switch.
- **Exception handling**: Code to run when exceptions arise.

Base and bounds virtualization is quite *efficient* but it has a **lot** of problems. As can be seen in Figure 3.2 all of the space between the stack and heap is wasted. This is called **internal fragmentation**.

3.2.2 Segmentation

The Crux: How To Support A Large Address Space

How do we support a large address space with (potentially) a lot of free space between the stack and heap?

Basic idea → **Segmentation**. Instead of having just one base and bounds pair in our MMU, why not have a base and bounds pair per logical segment of the address space? In our address space, we have three logically-different segments: code, stack, and heap. In segmentation, we place each one of those segments in different parts of memory.

Which Segment Are We Referring To?

Two approaches:

1. **Explicit** approach: Chop up the address space into segments based on few bits of the virtual address.

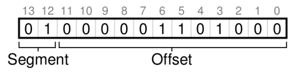


Figure 3.3: Chopping virtual address

2. **Implicity** approach: The hardware determines the segment by noticing how the address was formed.

Since out memory can grow forward or backward, we add another piece of information in hardware that wether the segment grows positively or negatively.

Support for Sharing

To save memory, sometimes it is useful to **share** certain memory segments between address spaces. In particular, **code sharing** is common nowdays.

To support this, we need to add more support for hardware in form of **protection bits**. Basic support adds a few bits per segment, indicating read-/write/execute permissions. Trying to do something out of permission should raise an exception.

OS Support

1. The OS has to save all base and bound register when a context switch occurs.

2. OS interaction when segments grow: if malloc() os called, in some cases the heap may be able to service the object. If it can, find free space for the object and return a pointer to it. If the heap needs to grow, call a system call to grow the heap.

3. Managing free space: When a new address is created, the OS has to be able to find space in physical memory for its segments. The general problem that arises is that physical memory soon becomes full of holes and to allocate a contigous block of memory, you will have to move some blocks around. This problem is called **external fragmentation**.

The main problem right now is using a varibable sized chunks.

3.3 Paging

Chop up space into *fixed-size* pieces → **paging**

The Curx: How To Virtualize Memory With Pages
How can we virtualize memory with pages, so as to avoid the problems faced by segmentation? What are the techniques and how do we make them work efficently?

Paging, has a number of advantages over our previous approaches:

- Probably the most important improvement will be *flexibility*: with a fully-developed pagin approach, the system will be able to support the abstraction of an address space effectively, regardless of how a process uses the address space. For example, no assumptions in the way stack and heap grow.
- Another advantage is the *simplicity* of free-space management that paging affords. For example, when the OS wishes to allocate lets say 64-byte address space into the physical memory, it just finds four free pages for this. OS keeps some kind of **free list** of all free pages for this.

To record where each virtual page of the address space is placed in physical memory, the operating system usually keeps a *per-process* data structure known as **page table**. The major role of the page table is to store **address translations** for each of the virtual pages of the address space, thus letting us know where in physical memory each page resides.

The page table is a *per-process* data structure (most page table structures we do here are like this, an exception is **inverted page table**). If another process wanted to run, we would have to change our page table.

3.3.1 Virtual Address Translation

Let's imagine the process with a tiny address space (64 bytes) is performing a memory access:

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

To **translate** this virtual address that the process generated, we have to first split it into two components: the **virtual page number (VPN)**, abd the **offset** within this page. For this example, we need need 6 bits total as $2^6 = 64$. Because we know the page size (16 bytes), we divide the virtual address as follows:

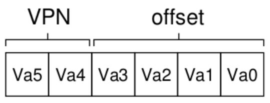


Figure 3.4: Virtual Address of the example

Lets say that we have out virtual address as 21, the VPN is 01, lets say it maps to the 7^{th} **physical frame number (PFN)** also called the **physical page number (PPM)**. Thus we can translate this address by replacing the VPN with the PFN and then issue the load to physical memory.

Note that the offset stays the same, because the offset just tells us which byte *within* the page we want.

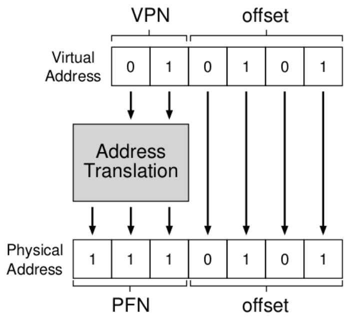


Figure 3.5: The Address Translation Process

3.3.2 Where Are Page Tables Stored?

Page tables can get terribly large, much bigger than the small segment table or base/bounds pair we discussed previously. For example, for a 32-bit address space, with 4KB pages, we have a 20-bit VPN and 12-bit offset. A 20-bit VPN implies that there are 2^20 translations that the OS would have to manage for each process. Assuming we need 4 bytes per **page table entry (PTE)** to hold the physical translation plus any other useful info, we get 4MB of memory needed for each page table, i.e. we need 4MB for each process. If we have a 100 processes running at a time, the Page Tables are using 400MB, which is too much. This will be even worse for 64-bit address space. Since they are so big, we dont keep any special onchip-hardware in MMU to store. We store it in the memory.

3.3.3 What is in the Page Table?

The page table is just a data structure used to map virtual addresses to physical addresses. For now, we use a **linear page table**, which is just an array. Later we'll use more complex structures to reduce memory.

We add the following things to a PTE:

1. A **valid bit** to indicate if the particular translation is valid. For example, on a running code we have stack and heap on opposite ends. The space inbetween them will be invalid. Accessing invalid positions will generate a trap to OS which will say GOODBYE OFFENDING PROCESS.
2. **protection bits** indicating read/write/execute permissions.
3. A **present bit** indicates wether this page is **swapped out** (Will study this further)
4. A **dirty bit** indicating if the page has been modified since the last time it was brought into memory.

5. **reference bit** Useful in determining which pages are popular. Such knowledge is critical during **page replacement** (Further)

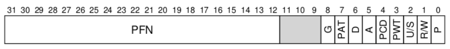


Figure 3.6: An x86 Page Table Entry (PTE)

3.3.4 Paging: Also Too Slow

The OS first goes to the memory and looks at the page table, brings back the translation, then again goes to memory to access the physical address mapped to the virtual address. Here is code for how it is done:

```
// 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 3.7: Accessing Memory with Paging

We have extra memory references which likely slow down the process by a factor of two or more.

Problem with paging → if not designed carefully, it will cause the system to run too slow as well as take too much memory. We need to overcome these two problems now.

3.3.5 Faster Translations (TLBs)

The Crux: How To Speed Up Address Translation

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

To speed up address translation, we are going to add **translation- lookaside buffer or TLB**. A TLB is part of the chip's **memory- management unit (MMU)**. It is basically **cache** for popular virtual-to-physical translations.

The algorithm hardware follows is like this:

- Extract the VPN from virtual address
- If the VPN is present in the TLB then it is a **TLB hit**. Extract the PFN and concatenate it with the offset.
- If it is a **TLB miss**, hardware accesses the page table to find the translation, assuming that the virtual reference generated by the process is valid and accessible, updates the TLB with the translation. A TLB miss is expensive because of the extra memory reference.
- Finally, once the translation is found in the TLB, the memory reference is processed quickly.

The TLB like all caches is based on spatial and temporal locality.

Who Handles The TLB Miss?

TLB miss can be handled by the hardware or the OS.

In the olden days, the hardware had **CISC** architecutre and handled the TLB miss entirely. To do this, the hardware has to know exactly *where* the page tables are located in memory as well as their *exact format*. On a miss, the hardware would find the page-table entry and extract the desired translation, update the TLB with the translation and retry the instruction. Intel x86 architecture uses **hardware-managed page TLBs** with a fixed **multi-level page table** (will study further)

More modern architectures with **RISC** style have what is known as a **software-managed TLB**. On a TLB miss, the hardware simple raises an exception, which pauses the current instruction stream, raises the privilege level to kernal mode and jumps to a **trap handler**. The trap handler handles the miss. Main reason of using OS to handle miss is more *flexibility*.

TLB Contents

A TLB has working similar to cache. But what is the data in it? A TLB entry might look like this:

VPN | PFN | other bits

The interesting part is the "other bits". For example, the TLB commonly has a **valid** bit, which says wether the entry has a valid translation or not. Also common are **protection** bits, which determine the permissions the process has for the memory. There might be other fields, including an **address-space indentifier**, a **dirty bit**, and so forth.

Context switches with TLB

The Crux: How To Manage TLB Contents On A Context Switch

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

One way would be to flush the TLB on context switches, thus emptying it before running the new process. However this is costly if the processes are switched frequently.

To reduce this overhead, some systems add hardware support to enable sharing of the TLB across context switches. In particular some systems provide an **address space identifier (ASID)** field in the TLB which can be thought of as a **process indentifier (PID)**, but usually takes fewer bits. Here is a depiction of TLB with the added ASID field:

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

Figure 3.8: TLD entry with ASID

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

Replacement Policy

The replacement policies are same as **cache replacement**. Duh, its a cache. What were you expecting? Anyway, this part is covered in swapping.

3.3.6 Paging: Smaller Tables

The Crux: How To Make Page Tables Smaller

Simple array-based page tables are too big. How can we make page tables smaller? What are the key ideas? What inefficiencies arise as a result of these new data structures?

Some solutions with their problems:

- **Bigger Pages:** Can lead to internal fragmentation due to pages not being used completly.
- **Paging + Segments**: Divide segments into pages. Doesnt really solve the main problem of segmentation i.e. external fragmentation.

Multi-level Page Tables

Get rid of all the invalid regions in the page table instead of keeping them all in memory.

We turn the linear page table into something like a tree. The basic idea is simple, chop up the apge table into page-sized units; then if an entire page of page-table entries (PTEs) is invalid, dont allocate that page of the page table at all. To track wether a page of the page table is valid, use a new structure called the **page directory**. the page tells you where a page of the page table is or that the entire page of the page table contains no vali pages.

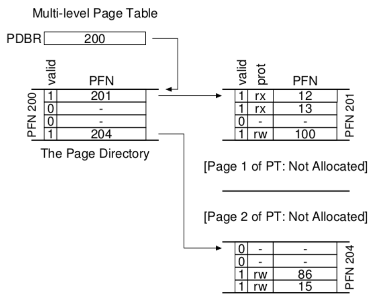


Figure 3.9: 2-Level Page Table

Advantages of Multi-level page tables:

- They only allocate page-table space in proportion to the amount of address space you are using; thus it is good for sparse address spaces

- If carefully constructed, each portion fits neatly within a page, making it easier to manage memory.

It should be noted that there is a cost to multi-level tables; for the 2-level page table, we required two loads from memory to get the right translation information from the page table (one for the page directory and one for the PTE itself). Smaller tables give higher TLB miss cost but reduce space. This is a space-time tradeoff. The cost in a TLB hit is the same.

We dont have to use only 2-level page tables. For larger physical memory, we need to use higher level page tables to get smaller tables. We create a page directory of page directories and so on. Levels can be decided based on how much reduction we need.

New code for Address translation:

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

Figure 3.10: Mutli-level Page Table Control Flow

As it can be seen, only the cost of TLB miss has been increased and not that of a TLB miss.

3.4 Swapping

Now we relax the assumption that our address space is unrealistically small and fits into physical memory,

The Crux: How To Go Beyond Physical Memory

How can the OS make use of a larger, slower device to transparently provide the illusion of a large virtual address space?

First, we need to reserve some space for swaping. we generally refer to such space as **swap space**. Thus, we simply assume that the OS can read from and write to the swap space, in page-sized units. OS needs to remember the **disk address** of a given page to accomplish this.

Now that we have some space on disk for swapping, we need to add machinery in the system to support swapping pages to and from the disk. We assume that we have a system with a hardware-managed TLB (like x86 has).

We need to add more machinery to the **page table entry (PTE)**. It must now have a **preset bit**, which indicates wether the page is in memory. In the case that the hardware looks in the TLB and the page is *not present*, the OS is invoked to service the **page fault**. The handler invoked is called **page-fault handler**. It runs and services the page fault.

When the OS receives a page fault for a page, it looks in the PTE to find the address, and request to disk to fetch page into memory. When disk I/O completes , the OS will then update the page table to mark the page as present, update the PFN field of the page-entry (PTE) to record the in-memory location of the newly-fetched page, and retry the instruction. While the I/O is in flight, the process will be blocked and the OS is free to execute other processes.

In the process above, if the memory is full then we use a **page replacement policy** to kick out, or replace pages from memory.

Chapter 4

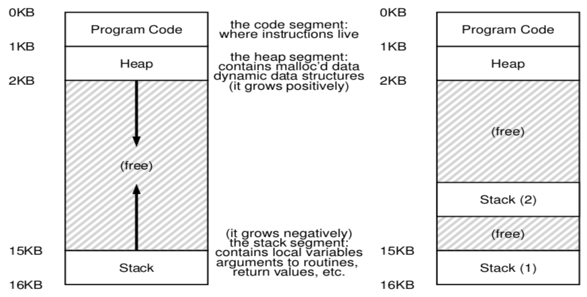
Concurrency

4.1 Introduction

In this chapter, we introduce a new abstraction for a single running process: that of a **thread**. Each thread is very much like a separate process, except for one difference: they share the same address space and thus can access the same data.

The state of a single thread is very similar to that of a process. It has a program counter, a private set of registers. Two switch between two different threads, a **context switch** must occur. With processes we saved state to a process control block (PCB); now, we need one or more **thread control blocks (TCBs)**.

In multi-threaded processes, there are multiple stacks in a single process. This is thread local storage. This is destroyed after thread is finished. Better not to allocated memory here which is meant to be used later.



0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack
the code segment: where instructions live
the heap segment: contains malloc'd data dynamic data structures (it grows positively)
(it grows negatively) the stack segment: contains local variables arguments to routines, return values, etc.
0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack (2)
(free)
Stack (1)

Figure 4.1: Single vs Multi Threaded Address Spaces

Why use threads?

1. **Parallelism:** Modern CPUs are multi-core. To take advantage of this fact.

2. **Avoid blocking:** One thread does I/O while others does work.

4.1.1 Problem: Shared Data

Lets take a simple example where two threads wish to update a single global shared variable.

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

static volatile int counter = 0;

// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n",
            counter);
    return 0;
}
```

Figure 4.2: Sharing Data Problem

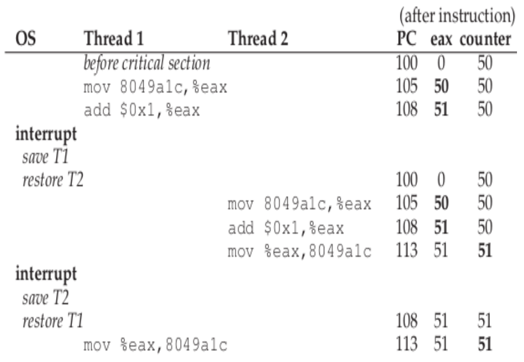
(Here functions of pthread library with 'P' are same as with small 'p' with some error checking)

When we run this code, we observe weird behavior. We get a wrong and weird result each time.

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

Figure 4.3: Wrong result

The image below depicts why this happens:



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

Figure 4.4: The Problem

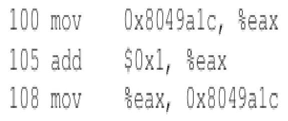
The problem demonstrated here is call a **race condition**. Each thread tries to update the counter. The one that updates it later, actually updates it. Because multiple threads executing the small piece of code can cause a race condition, this piece is called a **critical section**. A critical section is a piece of code that accesses a shared variable (or more generally, a shared resource) and must not be concurrently executed by more than one thread.

What we want from this code is **mutual exclusion**. This propery gurrantees that only one thread will be in the critical section.

Atmoic Operation: A series of actions simply depicted by the sentence "All or nothing". An atomic operation is a powerful instruction in hardware which is or appears to be excuted at once on the cpu.

4.1.2 Wish for Atomicity

What we require to remove the data race in the problem above is that we want the following instructions to be an atomic instruction:



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

Figure 4.5: Need for atomic instruction

If we had a single instruction to do this, we could just use that instruction. But in general case, we wont have a single instruction to build a concurrent B-tree. Thus, we will use hardware support to get some **synchronization primitives**.

The Crux: How to Support Synchronization

What support does the hardware need to provide in order to build useful synchronization primitives? What support do we need from OS? How can we build primitives correctly and efficently.

4.1.3 Another Problem: Waiting for Another

Another common problem that arrises in concurrency is that of waiting for another thread while it finishes. Thus concurrency is not only about making shared resouces accessable but also about this waking/sleeping interaction.

4.2 Locks

We saw that one of the fundamental problems in concurrent programming: we would like to execute a series of instructions atomically, but due to the presence of interrupts on a single processor, we couldn't. Thus, we introduce **locks**.

We wrap a critical section around a critical section and it then executes it as if it were a single atomic instruction.

4.2.1 Basic Idea

Lets assume our critical section is this:

$$balance = balance + 1;$$

Other critical sections are possible, but when we genearlize the principle, it does not matter. To use a lock, we add it around the critical section as shown in 4.2.1

A lock is just a shared variable (must be initialized) that keeps track if the critical section can be entered by the running thread. The lock variable or "lock" for short, holds the state of lock at any time.

Calling $lock()$ tries to acquire the lock. If no other thread has the lock, the lock is acquired and the thread enters the critical section. If another thread calls $lock()$ on the same lock variable, it will have to wait for the other thread to release the lock first.

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

Figure 4.6: Using a lock

Once the owner of the lock calls *unlock*(), the lock is free again. If there are waiting threads, one of the will acquire the lock.

Pthread Locks

The name that the POSIX library uses for a lock is a **mutex**, as it provides mutual exclusion between threads. Here is how you use it:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock); // wrapper; exits on failure
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

Figure 4.7: Using a POSIX lock

As it can be seen, POSIX provides **fine-grained** approach i.e. allows different locks insted of one big lock over the critical section i.e. **course-grained** locking stratergy.

4.2.2 Building A Lock

The Crux: How To Build A Lock

How can we build an efficent lock? What hardware support is needed? What OS support is needed?

Evaluating a Lock

1. Correctness: Does it provide mutual exclusion?

2. Fairness: Is the lock fair? i.e. Does each thread contending for the lock get and equal shot at acquiring it?

3. Performance: How bad is the time overhead of using the lock

Controlling Interrupts

An early solution to provide mutual exclusion was to disable interrupts for critical sections of code; this solution was for single-processor systems.

The problem with this approach was that disablling interrupts is a *privileged* operation. This way is not secure as the user program can take control by running an infinite loop inside the lock.

Another problem is that this approach is that it does not work on multi-processor systems.

This approach is generally avoided. It is sometimes used by the OS to handle is own kernal-based data structures. Trust issue is not a problem here as the OS trusts itself.

Just Using Loads/Stores

Let's try building a simple lock by using a single flag variable:

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

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

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

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

Figure 4.8: A Simple Flag

This code has two problems.

1. Correctness: This (1) code does not provide mutual exclusion

2. Performance: A thread waiting to acquire a lock which is already held endlessly checks the value of the flag in its time slice. This is known as **spin-waiting**. This wastes a lot of time while another thread could be running on the processor.

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 4.9: No Mutual Exclusion

Working Sping Locks with Test-And-Set

Today, all systems provide support for an atmoic intsruction to do a **test-and-set** (or **atmoic exchange**). This (4.2.2) instruction does this atomically.

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

Figure 4.10: Test and Set

This **atmoic** instruction is enough to build a **spin lock** as shown in 4.2.2

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

void init(lock_t *lock) {
    // 0: lock is available, 1: lock 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 4.11: A Working Spin Lock

Evaluating Spin Locks

1. Correctness: Spin locks do provide mutual exclusion.

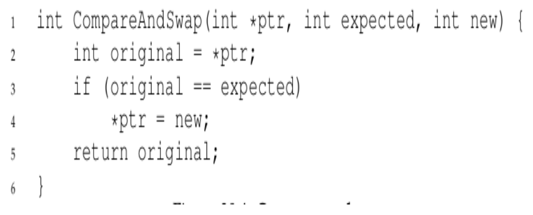
2. Fairness: We cannot guarantee that a waiting thread will ever enter the critical section. Spin locks do not provide any fairness gurantees.

3. Performance: Performance overhead can be very bad due to spin-waiting. Imagine the case where N threads compete for a lock and one of them holds the lock. Now, lets say the schedular runs each of the $N-1$ threads and tries to acquire the lock and thus waiting and wasting their time slice. This is very bad for performance.

   On multi processor systems, this isnt as bad assuming that number of threads is less than number of available CPUs because usually the critical section is very small and a thread running on another CPU can acquire it as soon as the owner of lock releases it.

Using Compare and Swap

Another hardware primitive that some systems provide is the **compare-and-swap** instruction. The C psudeocode for the instruction:



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

Figure 4.12: Compare and Swap

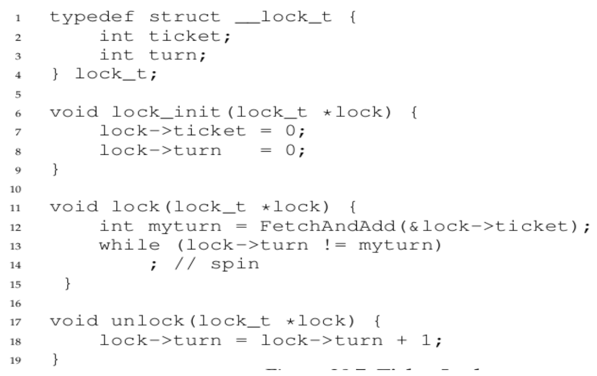
To make a spin-lock using compare-and-swap, we can just replace the test-and-swap instructon and modify a bit to do the same thing.

Fetch And Add

Another hardware primitive is the **fetch-and-add** instruction, which atomically increments a value and returns it. Using this instruction we can make a more interesting lock called a **ticket lock**.

Instead of a single value, this solution uses a ticket and turn variable in combination to build a lock. The operation is simple: whn a thread wishes to acquire the lock, it first does an atomic fetch-and-add on the ticket value; that value is now considered this thread's "turn". The globally shared lock-¿turn is then used to determine which thread's turn it is for a given thread.

The important difference between ticket locks and spin locks is that ticket locks guarantee fairness while spin locks don't. Each threads progresses. We can solve the problem of fairness using ticket locks. But they still have the problem of spinning too much.



```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn   = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
 }

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

Figure 4.13: Ticket Locks

4.2.3 Reducing the amoung of Spining

This part is related to making the locks more efficent by reducing the amount of spining that they do.

The Crux: How to Avoid Spining

How can we develop a lock that doesn't needlessly waste time spining on the CPU.

We need OS support for this now.

Using Yield

In this approach we assume that the OS provides a primitive *yield*() which a thread can call when it wants to give up the CPU and let another thread run. A thread can be in one of thethree states (running, read, or blocked); yield is simply a system call that moves the caller from the **runnning** state to the **ready** state. Thus allowing another thread to run. The yielding thread essentially **descedules** itself.

This approach still has a problem of fairness and that of efficency as this approach leaves it all to the schedular which may do anything.

Using Queues: Sleeping Instead of Spining

The problem with previous approaches was they leave too much to chance (i.e. behavior) of the schedular. This is not optimal as we have worst cases which can occur easily. Thus, we must explicitly exert some control over which threads acquires the lock after the current holder releases it.

We use an OS primitive: *park*() which puts the calling thread to sleep and *unpark*() to wake a particular thread. Using this we can build a lock which solves our problems as shown in 4.2.3.

```
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag  = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}

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

Figure 4.14: Locks with queues, test-and-set, yield, and wakeup

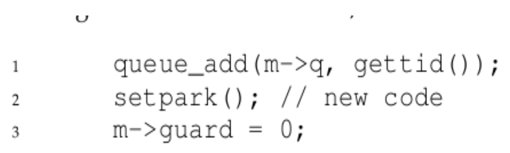
We combine the old test-and-set idea with an explicit queue of lock waiters to make a more efficent lock. Second, we use a queue to help control who gets the lock next and thus avoid starvation.

This approach does not avoid spin-waiting; However the time spent spinning is quite limited, and thus this approach may be reasonable.

There is one problem with this approach. there is a race conditon in the solution, just before the call to *park*(). With the wrong timing, a thread will be about to park, assuming that it should sleep until the lock is no longer held.

A switch at that time to another thread, (say a thread holding the lock) coud lead to trouble, for example the owner of the lock releases the lock and tries to unpark this thread and then the thread runs and parks itself. It will now go into a sleep forever.

This problem can be solved by another OS primitive called *setpark*(). By calling this routine, a thread can indicate it is *about* to park. If it then happens to be interrupted and another thread calls unpark before park is actuall called, the subsquent park returns immediately instead of sleeping. The code modification, inside the *lock*(), is quite small as shown in 4.2.3



```
    queue_add(m->q, gettid());
    setpark(); // new code
    m->guard = 0;
```

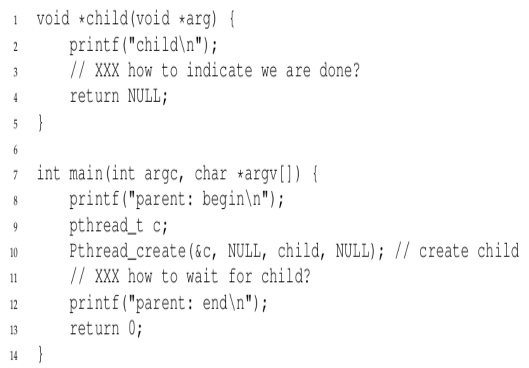
Figure 4.15: Using setpark

Note: *park*(), *unpark*(), *setpark*() are part of Solaris. Other OS may have different support.

4.3 Condition Variables

In many cases, a thread wishes to check whether a **condition** is true before continuing its execution. For example, a parent thread might wish to check wether a child thread has completed before continuing (often called a *join*()).

The problem is shown in 4.3



```
void *child(void *arg) {
    printf("child\n");
    // XXX how to indicate we are done?
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    // XXX how to wait for child?
    printf("parent: end\n");
    return 0;
}
```

Figure 4.16: A Parent waiting for its Child

We could try to use a shared variable and just spin. But this is very inefficent as the parent spins and wastes CPU time as shown in 4.3

```
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 4.17: Spin based approach

The Crux: How To Wait For A Condition

In multi-threaded programs, it is often useful for a thread to wait for some condition to become true before proceeding. How should a thread wait for a condition?

A **condition variable** is an explicit queue that threads can put themselves on when some state of execution (i.e., some **condition**) is not as desired (by **waiting** on the condition). Some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue (by **signaling** on the condition).

A thing to notice about *wait*() call is that it also takes a lock as a paramter. It assumes that this mutex is locked when *wait*() is called. The responsibility of *wait*() is to release the lock and put the calling thread to sleep (atomically); When the thread wakes up (after some thread has signaled it), it must re-acquire the lock before returning to the caller.

Two things to better understand the code:

1. Usage of the state variable *done*: Lets say we did not use this. The approach is broken then. The child may run and finish and signal immediatly after its created. And then the parent tries to wait and goes to a sleep forever.

```
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Figure 4.18: Using a condition variable to Solve Fork-Join problem

2. Usage of locks: This approach is again broken as after the parent checks for *done* == 0, the child may signal before the wait is executed and thus the parent goes in an infinite sleep.

Tip: Always hold the lock while signaling. A good general rule.

4.4 The Bounded Buffer Problem

Imagine one or more producer threads and one or more consumer threads. Producers generate data items and place them in a buffer; consumers grap said items from the buffer and consume them in some way. Because the bounded buffer is a shared resource, we require synchronized access to it.

Lets try to come up with a solution for only one item in the buffer.

First solution

With just a single consumer and single producer, this code works. But if we have more than one of these threads, the solution breaks.

```
int loops; // must initialize somewhere...
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);           // p1
        if (count == 1)                       // p2
            Pthread_cond_wait(&cond, &mutex); // p3
        put(i);                               // p4
        Pthread_cond_signal(&cond);           // p5
        Pthread_mutex_unlock(&mutex);         // p6
    }
}

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

Figure 4.19: Single CV and If Statement

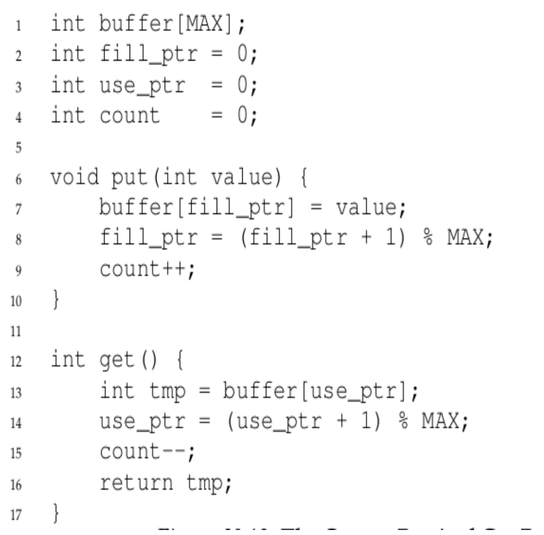
The problem is with the *if* statement. Lets say we have two running consumers T_{c_1}, T_{c_2} and one producer P. T_{c_1} is schedulded first and it sleeps. P is schedulded after it and it signals and now T_{c_1} is in *ready* queue. But now, T_{c_2} runs. T_{c_2} runs and finishes and now T_{c_1} runs and fails because there is nothing in the buffer to take but it still tries to acquire something.

This problem arrises because the OS does not gurrantee that the thread woken up will immediatly run. This reffered to as **Mesa semantics**. The opposite behavior is called **Hoare semantics**, which provide a harder guarantee that woken thread will run immediately upon being woken up.

Second solution

The previous problem is easily fixable by converting the *if* to a *while* statement. Generally, in Mesa Semantics, its **always better to use while loops**.

Another easy to see problem with this code is that there is only one condition variable and it may signal the wrong thread. To fix that, we use two condition variables instead.



```
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 4.20: The Correct Put and Get Routines

Full Producer/Consumer Solution

4.5 Semaphores

A semaphore is an object with an integer values that we can manipulate with two routines: *sem_wait*() and *sem_post*() (Called so in POSIX standard). A semaphore must be initialized to some value based on use (Initial value must be number of resources available at a time).

Figure 4.5 illustrates the use of these functions.

When the vale of semaphore is negative, it is equal to the number of waiting threads. This invariant is worth knowing.

Binary Semaphores (Locks)

To use the semaphore as a lock, its initial value should be 1. Now, *sem_wait*() tries to acquire the lock and *sem_post*() releases the lock.

Semaphores for Ordering

Semaphores can be used to wait for a procedure to complete before moving on to the next one as shown in 4.5

To use this kind of ordering the semaphore needs to be initialised to 0 as it is clearly visable how it works.

```
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 4.21: Producer/Consumer Synchronization

```
int sem_wait(sem_t *s) {
    decrement the value of semaphore s by one
    wait if value of semaphore s is negative
}

int sem_post(sem_t *s) {
    increment the value of semaphore s by one
    if there are one or more threads waiting, wake one
}
```

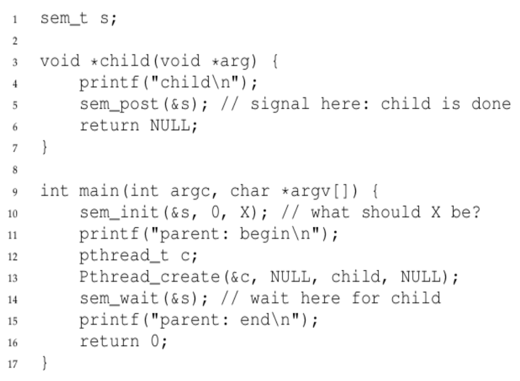
Figure 4.22: Wait and Post

4.5.1 Bounded Buffer Problem with Semaphores

First Attempt

First attempt just uses two semaphores as shown in 4.5.1

This solution works fine for single consumer and single produces. But lets say we have multiple consumers and multiple produces and queue of size greater than 1. Then there is a *data race* here. Two producers may try to call *put*() at the same time.

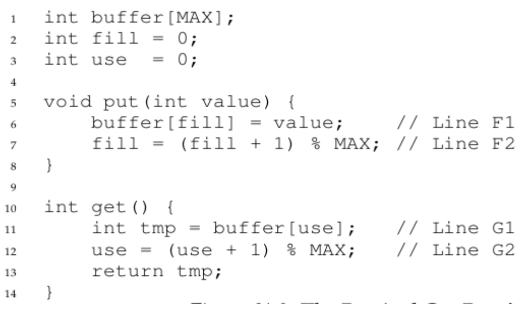


```
sem_t s;

void *child(void *arg) {
    printf("child\n");
    sem_post(&s); // signal here: child is done
    return NULL;
}

int main(int argc, char *argv[]) {
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL);
    sem_wait(&s); // wait here for child
    printf("parent: end\n");
    return 0;
}
```

Figure 4.23: Parent waiting for its Child



```
int buffer[MAX];
int fill = 0;
int use  = 0;

void put(int value) {
    buffer[fill] = value;    // Line F1
    fill = (fill + 1) % MAX; // Line F2
}

int get() {
    int tmp = buffer[use];   // Line G1
    use = (use + 1) % MAX;   // Line G2
    return tmp;
}
```

Figure 4.24: Put and Get Routines

Adding More Mutual Exclusion

Adding locks can remove the data race happening in 4.5.1 as shown in 4.5.1

One thing to note about this approach is that it causes a *deadlock*. Lets say a producers gets the lock while the queue is full, now the process is stuck in a deadlock as the lock cannot be released until queue isnt full and to remove something from the queue, the lock needs to be released.

Avoiding Deadlock

The working solution without deadlocks is given in 4.5.1

4.5.2 Reader-Writer Locks

Multiple reader threads can access a structure at once but when writing, only the writer thread should be inside the critical section. This type of lock can be implemented using semaphores as shown in **??**

```c
sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);       // Line P1
        put(i);                 // Line P2
        sem_post(&full);        // Line P3
    }
}

void *consumer(void *arg) {
    int tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);        // Line C1
        tmp = get();            // Line C2
        sem_post(&empty);       // Line C3
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX are empty
    sem_init(&full, 0, 0);    // 0 are full
    // ...
}
```

Figure 4.25: First attempt on Bounded Buffer

```c
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);       // Line P0 (NEW LINE)
        sem_wait(&empty);       // Line P1
        put(i);                 // Line P2
        sem_post(&full);        // Line P3
        sem_post(&mutex);       // Line P4 (NEW LINE)
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);       // Line C0 (NEW LINE)
        sem_wait(&full);        // Line C1
        int tmp = get();        // Line C2
        sem_post(&empty);       // Line C3
        sem_post(&mutex);       // Line C4 (NEW LINE)
        printf("%d\n", tmp);
    }
}
```

Figure 4.26: Adding Mutual Exclusion

4.5.3 Thread Throttling

How to put a threshold on the number of threads created at a time? Use semaphores to **throttle** the amount of threads at a time. Use *sem_wait*() around the thread creation region and *sem_post*() around the thread ending region.

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);       // Line P1
        sem_wait(&mutex);       // Line P1.5 (MUTEX HERE)
        put(i);                 // Line P2
        sem_post(&mutex);       // Line P2.5 (AND HERE)
        sem_post(&full);        // Line P3
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);        // Line C1
        sem_wait(&mutex);       // Line C1.5 (MUTEX HERE)
        int tmp = get();        // Line C2
        sem_post(&mutex);       // Line C2.5 (AND HERE)
        sem_post(&empty);       // Line C3
        printf("%d\n", tmp);
    }
}
```

Figure 4.27: Avoiding Deadlock

```
typedef struct _rwlock_t {
  sem_t lock;      // binary semaphore (basic lock)
  sem_t writelock; // allow ONE writer/MANY readers
  int   readers;   // #readers in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
  rw->readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1) // first reader gets writelock
    sem_wait(&rw->writelock);
  sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0) // last reader lets it go
    sem_post(&rw->writelock);
  sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
}
```

Figure 4.28: Simple Reader-Writer Lock