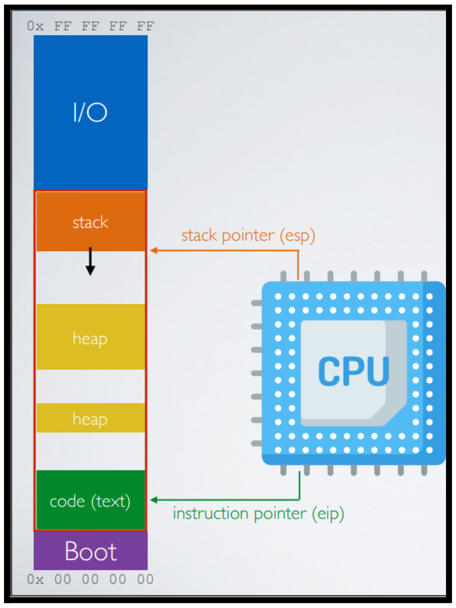
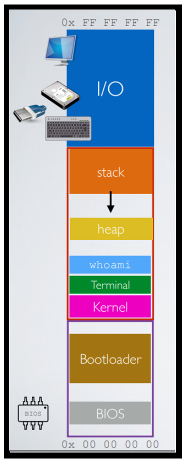
CSCC69 Week 1 Notes

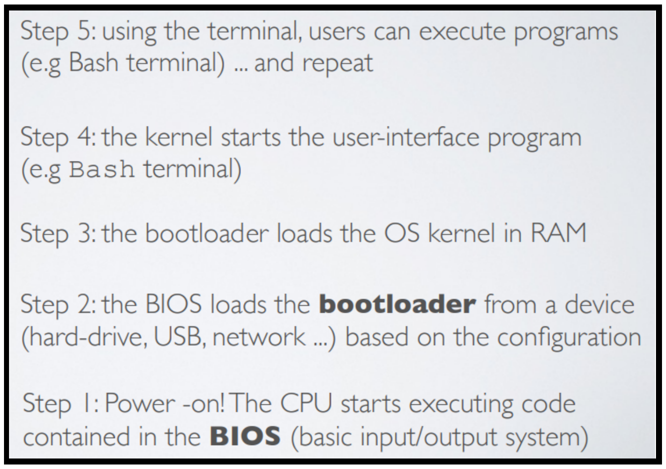
Lecture Notes:

- **Computer Architecture:**
- A simple computer architecture consists of Memory + CPU.
- A **CPU/Processor** is the logic circuitry that responds to and processes the basic instructions that drive a computer. The CPU is seen as the main and most crucial integrated circuit chip in a computer, as it is responsible for interpreting most computer commands. CPUs will perform most basic arithmetic, logic and I/O operations, as well as allocate commands for other chips and components running in a computer.
- Each processor has its **Instruction Set Architecture (ISA)**.
- The ISA provides commands to the processor, to tell it what it needs to do. It consists of addressing modes, instructions, native data types, registers, memory architecture, interrupt, and exception handling, and external I/O.
- Each instruction is a bit string that the processor understands as an operation:
    - arithmetic
    - read/write bit strings
    - bit logic
    - jumps
- There are about 2000 instructions on modern x86-64 processors.
- This is what happens when you run 1 program:



- A **stack** is a reserved area of memory used to keep track of a program's internal operations, including functions, return addresses, passed parameters, etc. It is the memory set aside as scratch space for a thread of execution. A stack is usually maintained as a "last in, first out" (LIFO) data structure, so that the last item added to the structure is the first item used.
- The **stack pointer** is used to indicate the location of the last item put onto the stack.
- The **heap** is memory set aside for dynamic allocation. I.e. When you use malloc, it goes on the heap. Unlike the stack, there's no enforced pattern to the allocation and deallocation of blocks from the heap; you can allocate a block at any time and free it at any time. This makes it much more complex to keep track of which parts of the heap are allocated or free at any given time.
- **Bootstrapping:**
- A **bootstrap program** is the first code that is executed when the computer system is started. It is the program that initializes the operating system during startup. The entire operating system depends on the bootstrap program to work correctly as it loads the operating system.
- The bootstrapping process does not require any outside input to start. Any software can be loaded as required by the operating system rather than loading all the software automatically. The bootstrapping process is performed as a chain.





Step 5: using the terminal, users can execute programs (e.g Bash terminal) ... and repeat

Step 4: the kernel starts the user-interface program (e.g `Bash` terminal)

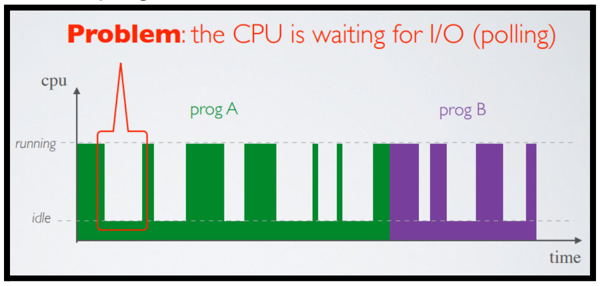
Step 3: the bootloader loads the OS kernel in RAM

Step 2: the BIOS loads the **bootloader** from a device (hard-drive, USB, network ...) based on the configuration

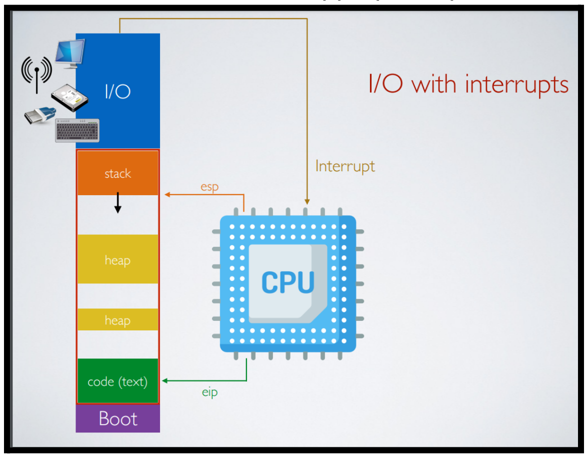
Step 1: Power -on! The CPU starts executing code contained in the **BIOS** (basic input/output system)

- Step 1: Power-on. The CPU starts executing code contained in the BIOS (basic input/output system).
  Step 2: The BIOS loads the bootloader from a device (hard-drive, USB, network, etc) based on the configuration.
  Step 3: The bootloader loads the OS kernel in RAM.
  Step 4: The kernel starts the user-interface program (e.g Bash terminal).
  Step 5: Using the terminal, users can execute programs (e.g Bash terminal) and repeat.
- **Concurrency:**
- **Concurrency** is the execution of the multiple instruction sequences at the same time. It happens in the operating system when there are several process threads running in parallel. The running process threads always communicate with each other through shared memory or message passing.
- Concurrency helps in techniques like coordinating execution of processes, memory allocation and execution scheduling for maximizing throughput.
- Problems in concurrency:
  - **Sharing global resources:** If two processes both make use of a global variable and both perform read and write on that variable, then the order in which various read and write are executed is critical.
  - **Optimal allocation of resources:** It is difficult for the operating system to manage the allocation of resources optimally.
  - **Locking the channel:** It may be inefficient for the operating system to simply lock the channel and prevent its use by other processes.
- Advantages of concurrency:
  - **Running of multiple applications:** Concurrency means that the OS can run multiple applications at the same time.
  - **Better resource utilization:** Concurrency enables that resources that are unused by one application can be used for other applications.
  - **Better average response time:** Without concurrency, each application has to be run to completion before the next one can be run.
  - **Better performance:** Concurrency enables better performance by the operating system. If one application only uses the processor and another application only uses the disk drive then the time to run both applications concurrently will be shorter than the time to run each application consecutively.
- Drawbacks of concurrency:
  - Coordinating multiple applications is difficult and complex.
  - Additional performance overheads and complexities in operating systems are required for switching among applications.
  - Sometimes running too many applications concurrently leads to severely degraded performance.

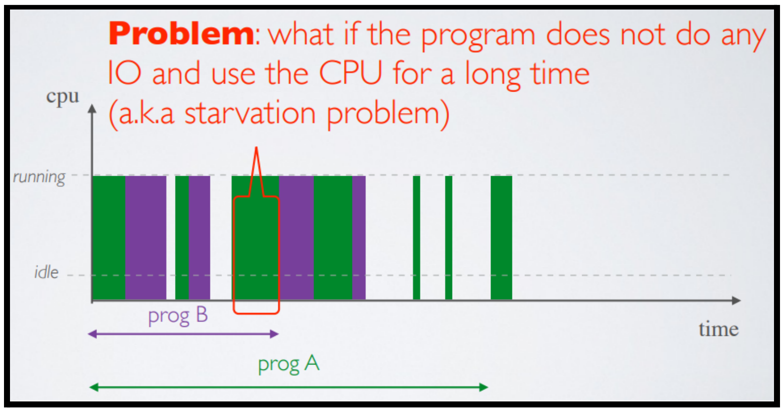
- Consider the following example: We are running 2 programs, A and B. If we run B after finishing running A, we're wasting a lot of time. This is because most programs have to deal with I/O, which is slow. So, we'll have a lot of time where we'll just be waiting for A to communicate with another program.



One way to solve this issue is to use **interrupts**. An interrupt is a signal emitted by hardware or software when a process or an event needs immediate attention. It alerts the processor to a high-priority process requiring interruption of the current working process. Interrupts are signals sent to the CPU by external devices, normally I/O devices. They tell the CPU to stop its current activities and execute the appropriate part of the operating system.



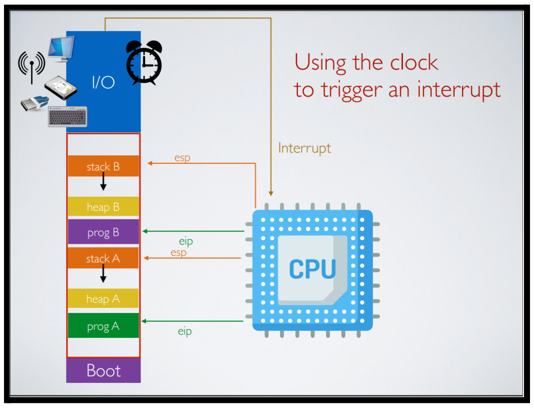
What happens is that while the CPU is waiting for one program, say A, to finish its communication, it will move on to another program. However, once A finishes its communication, it will generate an interrupt to let the CPU know it has finished its communication and to stop what it's currently doing and continue executing it. This way, while programs are in the middle of communicating (I/O), the CPU can work on other programs and won't have to waste time.



From the picture above, you can see that while A is in the middle of I/O, the CPU works on B.

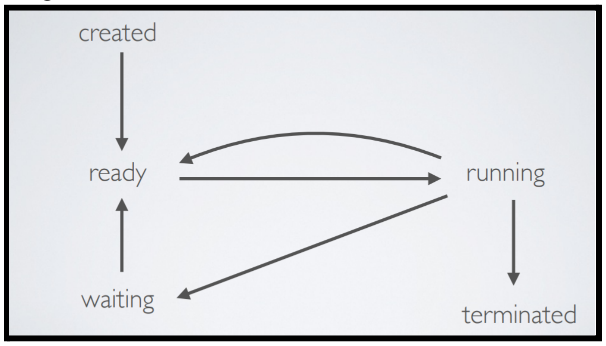
However, there are a few problems with interrupts:

1. Concurrent access to I/O devices must be synchronized. Suppose that both A and B need access to the same I/O device at the same time, what happens then?
2. What if the program does not do any IO and use the CPU for a long time?
   This is called the **starvation problem**.
   To fix this issue, we will use a system clock that triggers an interrupt at a specific time interval. We do this to avoid having the CPU stuck on one program for too long. Each time the system clock triggers an interrupt, the CPU will switch to another program.



   Note: The CPU is only executing (running) 1 program at a time.

- Program states:



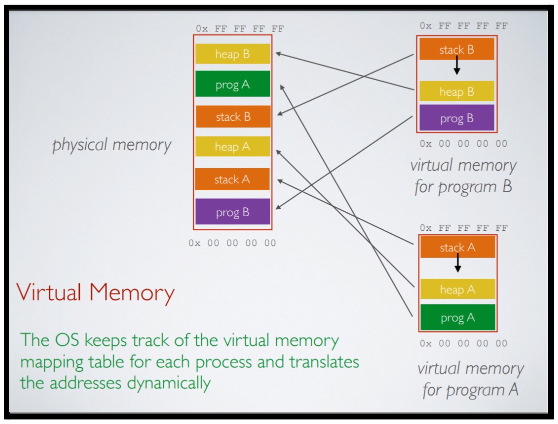
After a program is created, it will go to the **ready queue** to let the CPU know that it's ready. Then, at some point, the CPU will run that program. From there, the program has 3 options:

1. It is completely done running, so it is terminated.
2. It needs to wait for I/O, so it goes to waiting. After the I/O is complete, it will go back into the ready queue.
3. It is done running a section, generates an interrupt, and goes back to the ready queue.

- **Note:** Even though most modern computers/laptops have multiple cores, the same issues still apply since our computers/laptops are running hundreds of programs at a time. However, with multiple cores, interrupts become more difficult cause we have to decide which core will handle it.
- With concurrency:
    - From the system perspective, there is better CPU usage resulting in a faster execution overall but not individually.
    - From the user perspective programs seem to be executed in parallel.
    - But it requires scheduling, synchronization and some protection mechanisms.

- **User Programs:**
- Consider this example: You are writing a Bash shell that reads keyboard inputs from the user. However, there are many ways the user can send the keyboard inputs:
    1. The one connected to the USB (USB keyboard).
    2. The one connected to the bluetooth (Bluetooth keyboard).
    3. The remote one connected to the network (SSH).

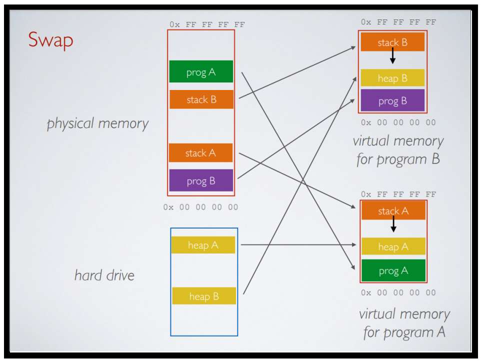
  How do you know which I/O device to listen to?
- User programs do not operate I/O devices directly.
- The OS abstracts those functionalities and provides them as **system calls**.
  I.e. The OS creates APIs for user programs to use so they don't interact with the kernel directly.
- A system call is a mechanism that provides the interface between a process and the operating system. It is a programmatic method in which a computer program requests a service from the kernel of the OS.
- System call offers the services of the operating system to the user programs via an API. System calls are the only entry points for the kernel system.
- System calls provide user programs with an API to use the services of the operating system.
- There are 6 categories of system calls:
    1. Process control
    2. File management
    3. Device management
    4. Information/maintenance (system configuration)
    5. Communication (IPC)
    6. Protection
- There are 393 system calls on Linux 3.7.
- **Virtual Memory:**
- A computer can address more memory than the amount physically installed on the system. This extra memory is called **virtual memory** and it is a section of a hard disk that's set up to emulate the computer's RAM. It is done by treating a part of secondary memory as the main memory. In virtual memory, the user can store processes with a bigger size than the available main memory.
- Therefore, instead of loading one long process in the main memory, the OS loads the various parts of more than one process in the main memory.
- The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using a disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.
- Consider this example: We want to make programs and execution contexts coexist in memory. Placing multiple execution contexts (stack and heap) at random locations in memory is not a problem as long as you have enough memory. However having programs placed at random locations is problematic. We can use virtual memory to solve this problem.



Virtual Memory

The OS keeps track of the virtual memory mapping table for each process and translates the addresses dynamically

- Another problem arises if we run out of memory because of too many concurrent programs. We can use virtual memory to solve this too. We can swap memory by moving some data to the disk. Managing memory becomes very complex but necessary.



- **File System:**
- A **file** is a named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tapes and optical disks. In general, a file is a sequence of bits, bytes, lines or records whose meaning is defined by the files creator and user.
- **Operating System:**
- In a nutshell, an OS:
    - Manages hardware and runs programs.
    - Creates and manages processes.
    - Manages access to the memory (including RAM and I/O).
    - Manages files and directories of the filesystem on disk(s).
    - Enforces protection mechanisms for reliability and security.
    - Enables inter-process communication.

Textbook Notes:

- **Introduction to Operating Systems:**
    - Introduction to Operating Systems:
    - The primary way the operating system (OS) makes it easy to seemingly run many at the same time, allows programs to share memory, and enables programs to interact with devices is through a general technique that we call **virtualization**. That is, the OS takes a physical resource (such as the processor, or memory, or a disk) and transforms it into a more general, powerful, and easy-to-use virtual form of itself. Thus, we sometimes refer to the operating system as a virtual machine.
    - In order to allow users to tell the OS what to do and thus make use of the features of the virtual machine, the OS also provides some APIs that you can call. A typical OS, in fact, exports a few hundred **system calls** that are available to applications. Because the OS provides these calls to run programs, access memory and devices, and other related actions, we also sometimes say that the OS provides a standard library to applications.
    - Finally, because virtualization allows many programs to run, and many programs to **concurrently** access their own instructions and data, and many programs to access devices, the OS is sometimes known as a resource manager. Each of the CPU, memory, and disk is a resource of the system; it is thus the operating system's role to manage those resources, doing so efficiently or fairly or indeed with many other possible goals in mind.
    - Virtualizing The CPU:
    - It turns out that the operating system, with some help from the hardware, creates the illusion that the system has a very large number of virtual CPUs. Turning a 1 or more CPU(s) into a seemingly infinite number of CPUs and thus allowing many programs to seemingly run at once is what we call **virtualizing the CPU**.
    - However, the ability to run multiple programs at once raises all sorts of new questions. For example, if two programs want to run at a particular time, which should run? This question is answered by a **policy** of the OS. Policies are used in many different places within an OS to answer these types of questions.

- <u>Virtualizing Memory:</u>
- The model of physical memory presented by modern machines is very simple. Memory is just an array of bytes; to read memory, one must specify an address to be able to access the data stored there. To write or update memory, one must also specify the data to be written to the given address.
- Memory is accessed all the time when a program is running. A program keeps all of its data structures in memory, and accesses them through various instructions, like loads and stores or other explicit instructions that access memory in doing their work. Don't forget that each instruction of the program is in memory too; thus memory is accessed on each instruction fetch.
- When the OS is virtualizing memory, each process has and accesses its own private **virtual address space**/**address space**, which the OS maps onto the physical memory of the machine. A memory reference within one running program does not affect the address space of other processes (or the OS itself); as far as the running program is concerned, it has physical memory all to itself. The reality, however, is that physical memory is a shared resource, managed by the operating system.
- <u>Persistence:</u>
- In system memory, data can be easily lost, such as when power goes away or the system crashes, any data in memory is lost. Thus, we need hardware and software to be able to store data persistently; such storage is thus critical to any system as users care a great deal about their data.
- The hardware comes in the form of some kind of input/output or I/O device; in modern systems, a hard drive is a common repository for long lived information, although solid-state drives are making headway in this arena as well.
- The software in the operating system that usually manages the disk is called the **file system**; it is thus responsible for storing any **files** the user creates in a reliable and efficient manner on the disks of the system.
- Unlike the abstractions provided by the OS for the CPU and memory, the OS does not create a private, virtualized disk for each application. Rather, it is assumed that oftentimes, users will want to share information that is in files.
- For performance reasons, most file systems first delay such writes for a while, hoping to batch them into larger groups.
- To handle the problems of system crashes during writes, most file systems incorporate some kind of intricate write protocol, such as **journaling** or **copy-on-write**, carefully ordering writes to disk to ensure that if a failure occurs during the write sequence, the system can recover to reasonable state afterwards.
- <u>Design Goals:</u>
- An OS takes physical resources, such as a CPU, memory, or disk, and virtualizes them, handles tough and tricky issues related to concurrency and stores files persistently, thus making them safe over the long-term.
- One of the most basic goals is to build up some abstractions in order to make the system convenient and easy to use.
- Another goal in designing and implementing an operating system is to provide high performance; another way to say this is our goal is to minimize the overheads of the OS. Virtualization and making the system easy to use are well worth it, but not at any cost; thus, we must strive to provide virtualization and other OS features without excessive overheads.
- Another goal will be to provide protection between applications, as well as between the OS and applications. Because we wish to allow many programs to run at the same time, we want to make sure that the malicious or accidental bad behavior of one does not harm others; we certainly don't want an application to be able to harm the OS itself. Protection is at the heart of one of the main principles underlying an operating system, which is that of isolation; isolating processes from one another is the key to protection and thus underlies much of what an OS must do.

    - The operating system must also run non-stop. When it fails, all applications running on the system fail as well. Because of this dependence, operating systems often strive to provide a high degree of reliability.
- **The Abstraction - The Process:**
    - Introduction:
    - A **process** is a running program.
    - The program itself is a lifeless thing: it just sits there on the disk, a bunch of instructions and maybe some static data, waiting to spring into action. It is the operating system that takes these bytes and gets them running, transforming the program into something useful.
    - The OS creates an illusion of having endless CPUs 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 only 1 or a few CPUs exist. This basic technique, known as **time sharing** of the CPU, allows users to run as many concurrent processes as they would like; the potential cost is performance, as each will run more slowly if the CPU(s) must be shared.
    - **Time sharing** is a basic technique used by an OS to share a resource. By allowing the resource to be used for a little while by one entity, and then a little while by another, and so forth, the resource in question can be shared by many. The counterpart of time sharing is **space sharing**, where a resource is divided in space among those who wish to use it. For example, disk space is naturally a spaceshared resource; once a block is assigned to a file, it is normally not assigned to another file until the user deletes the original file.
    - To implement virtualization of the CPU, and to implement it well, the OS will need both some low-level machinery and some high-level intelligence. We call the low-level machinery **mechanisms**. Mechanisms are low-level methods or protocols that implement a needed piece of functionality.
    - On top of these mechanisms resides some of the intelligence in the OS, in the form of **policies**. Policies are algorithms for making some kind of decision within the OS. For example, given a number of possible programs to run on a CPU, which program should the OS run? A scheduling policy in the OS will make this decision, likely using historical information, workload knowledge, and performance metrics to make its decision.
    - The Abstraction - A Process:
    - The abstraction provided by the OS of a running program is something we will call a **process**.
    - To understand what constitutes a process, we thus have to understand its **machine state**: what a program can read or update when it is running. At any given time, what parts of the machine are important to the execution of this program?
    - One obvious component of a machine state that comprises a process is its memory. Instructions lie in memory; the data that the running program reads and writes sits in memory as well. Thus the memory that the process can address (called its **address space**) is part of the process.
    - Also part of the process's machine state are **registers**; many instructions explicitly read or update registers and thus clearly they are important to the execution of the process. Note that there are some particularly special registers that form part of this machine state. For example, the program counter tells us which instruction of the program will execute next. Similarly a stack pointer and associated frame pointer are used to manage the stack for function parameters, local variables, and return addresses.
    - Finally, programs often access persistent storage devices too. Such I/O information might include a list of the files the process currently has open.
    - Process Creation - A Little More Detail:
    - The first thing that the OS must do to run a program is to load its code and any static data into memory, into the address space of the process. Programs initially reside on disk in some kind of executable format; thus, the process of loading a program and static data into memory requires the OS to read those bytes from disk and place them in memory somewhere.

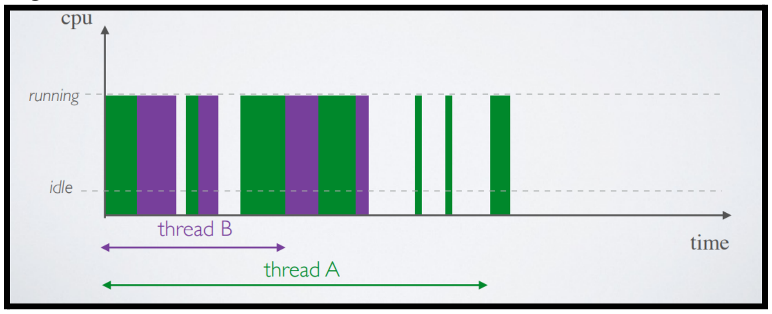
- In early operating systems, the loading process is done **eagerly** (All at once before running the program). Modern OSes perform the process **lazily** (by loading pieces of code or data only as they are needed during program execution).
- Once the code and static data are loaded into memory, there are a few other things the OS needs to do before running the process.
- Some memory must be allocated for the program's run-time stack.
- The OS may also allocate some memory for the program's heap. In C programs, the heap is used for explicitly requested dynamically-allocated data; programs request such space by calling malloc() and free it explicitly by calling free(). The heap will be small at first; as the program runs, and requests more memory via the malloc() library API, the OS may get involved and allocate more memory to the process to help satisfy such calls.
- The OS will also do some other initialization tasks, particularly as related to I/O. For example, in UNIX systems, each process by default has three open file descriptors, for standard input, output, and error. These descriptors let programs easily read input from the terminal and print output to the screen.
- By loading the code and static data into memory, by creating and initializing a stack, and by doing other work as related to I/O setup, the OS has now finally set the stage for program execution. It thus has one last task: to start the program running at the entry point, namely main(). By jumping to the main() routine, the OS transfers control of the CPU to the newly-created process, and thus the program begins its execution.
- <u>Process States:</u>
- There are 3 **states** a process can be in:
    1. **Running:** In the running state, a process is running on a processor. This means it is executing instructions.
    2. **Ready:** In the ready state, a process is ready to run but for some reason the OS has chosen not to run it at this given moment.
    3. **Blocked/Waiting:** In the blocked state, a process has performed some kind of operation that makes it not ready to run until some other event takes place. A common example: when a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.
- A process can be moved between the ready and running states at the discretion of the OS. Being moved from ready to running means the process has been **scheduled**. Being moved from running to ready means the process has been **descheduled**. Once a process has become blocked, the OS will keep it as such until some event occurs. At that point, the process moves to the ready state again.
- <u>Data Structures:</u>
- The OS is a program, and like any program, it has some key data structures that track various relevant pieces of information. To track the state of each process, for example, the OS likely will keep some kind of process list for all processes that are ready and some additional information to track which process is currently running. The OS must also track, in some way, blocked processes; when an I/O event completes, the OS should make sure to wake the correct process and ready it to run again.
- The **register context** will hold, for a stopped process, the contents of its registers. When a process is stopped, its registers will be saved to this memory location. By restoring these registers, the OS can resume running the process.
- Sometimes a system will have an **initial state** that the process is in when it is being created. Also, a process could be placed in a **final state** where it has exited but has not yet been cleaned up, sometimes called a **zombie state**.

- **The Abstraction - Address Spaces:**
    - The Address Space:
    - An **address space** is the running program's view of memory in the system. The address space of a process contains all of the memory state of the running program.
    - **Note:** When we describe the address space, what we are describing is the abstraction that the OS is providing to the running program.
    - One question/problem is: "How can the OS build this abstraction of a private, potentially large address space for multiple running processes all sharing memory on top of a single, physical memory?"

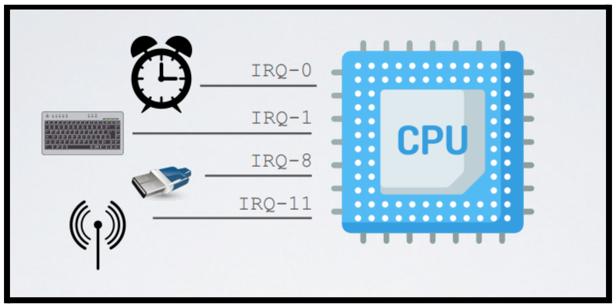
      The OS uses **virtual memory** to solve this problem. Virtual memory is a section of a hard disk that's set up to emulate the computer's RAM. It is done by treating a part of secondary memory as the main memory. In virtual memory, the user can store processes with a bigger size than the available main memory.
    - Goals:
    - One major goal of a virtual memory system is **transparency**. The OS should implement virtual memory in a way that is invisible to the running program. Thus, the program shouldn't be aware of the fact that memory is virtualized; rather, the program behaves as if it has its own private physical memory. Behind the scenes, the OS and hardware does all the work to multiplex memory among many different jobs, and hence implements the illusion.
    - Another goal of VM is **efficiency**. The OS should strive to make the virtualization as efficient as possible, both in terms of time and space.
    - Finally, a third VM goal is **protection**. The OS should make sure to protect processes from one another as well as the OS itself from processes. When one process performs a load, a store, or an instruction fetch, it should not be able to access or affect in any way the memory contents of any other process or the OS itself. Protection thus enables us to deliver the property of isolation among processes; each process should be running in its own isolated cocoon, safe from the ravages of other faulty or even malicious processes. **Isolation** is a key principle in building reliable systems. If two entities are properly isolated from one another, this implies that one can fail without affecting the other. Operating systems strive to isolate processes from each other and in this way prevent one from harming the other. By using memory isolation, the OS further ensures that running programs cannot affect the operation of the underlying OS.

<u>**Lecture Notes:**</u>

- **Program vs Thread:**
- **Program:** Static data on some storage.
- **Thread:** An instance of a program execution.
- Different threads executing the same program can run concurrently.
- **Running threads concurrently:**
- A CPU core will run multiple threads concurrently by running each thread for a little amount of time before switching to another one.
- There is limited direct execution. The CPU will switch to another thread when either the running thread yields the CPU (non-blocking IO for instance) or the CPU stops the running thread (system clock interrupt).
  E.g.

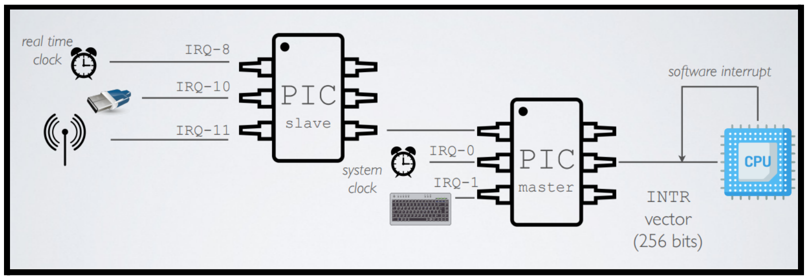


- Advantages of concurrency:
    - From the system perspective, there is better CPU usage resulting in a faster execution overall but not individually.
    - From the user perspective programs seem to be executed in parallel.
- **Interrupts:**
- There are 2 types of interrupts:
    1. **External Interrupts/Hardware Interrupts**. These are caused by an I/O device that needs some attention (asynchronous).
    2. **Internal Interrupts/System calls, exceptions and faults**. These are caused by executing instructions that have faults, such as dividing by 0 or page fault (synchronous).
- For hardware/external interrupts, this is the naive approach:



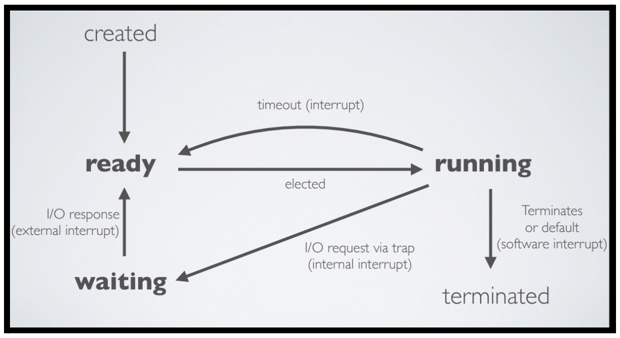
  Here, I/O devices are wired to **Interrupt Request lines** (IRQs). This means that it's not flexible (hardwired) and that the CPU might get interrupted all the time.

- A better approach is this:



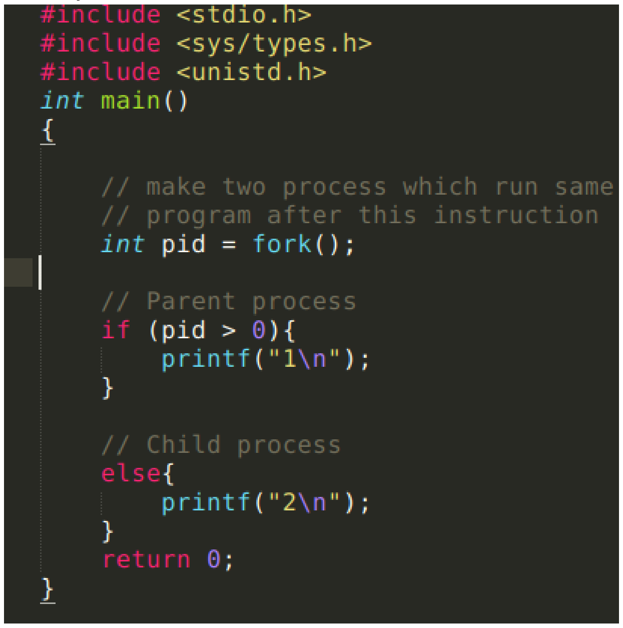
Here, I/O devices have unique or shared IRQs that are managed by two **Programmable Interrupt Controllers** (PIC).
- The PIC is responsible to tell the CPU when and which devices wish to interrupt through the INTR vector. There are 16 lines of interrupt (IRQ0 - IRQ15), interrupts have different priority and interrupts can be masked.
- Here are the steps for handling an interrupt:
    1. The CPU receives an interrupt on the INTR vector
    2. The CPU stops the running program and transfers control to the corresponding handler in the **Interrupt Descriptor Table** (IDT)
    3. The handler saves the current running program state
    4. The handler executes the functionality
    5. The handler restores or halt the running program
- These interrupt handlers are defined in:
    - Linux: cat /proc/interrupt
    - Windows: msinfo32.exe
- E.g. When a key is pressed:
    1. The keyboard controller tells PIC to cause an interrupt on IRQ #1
    2. The PIC, which decides if the CPU should be notified
    3. If so, IRQ 1 is translated into a vector number to index into CPU's Interrupt Descriptor Table
    4. The CPU stops the current running program
    5. The CPU invokes the current handler
    6. The handler talks to the keyboard controller via IN and OUT instructions to ask what key was pressed
    7. The handler does something with the result (e.g write to a file in Linux)
    8. The handler restores the running program
- **Context Switching:**
- A **context switch** is the process of storing the state of a process or thread, so that it can be restored and resume execution at a later point. This allows multiple processes to share a single (CPU), and is an essential feature of a multitasking operating system.
- When the CPU runs threads concurrently:
    - Only one thread at a time is running (on one core).
    - Several threads might be ready to be executed.
    - Several threads might be waiting for an I/O response.
- For each thread, the OS needs to keep track of its state (ready, running, waiting) and its execution context (registers, stack, heap and so on).

- The different states of a thread:


created
timeout (interrupt)
ready
running
elected
I/O response (external interrupt)
Terminates or default (software interrupt)
I/O request via trap (internal interrupt)
waiting
terminated

- Context switching occurs when:
    a. When the OS receives a fault:
        1. It suspends the execution of the running thread.
        2. It terminates the thread.
    b. When the OS receives a System Clock Interrupt or a System Call Trap (I/O request):
        1. It suspends the execution of the running thread.
        2. It saves its execution context.
        3. It changes the thread's state to ready (timeout) or waiting (I/O request).
        4. It elects a new thread from the ones in the ready state.
        5. It changes its state to running.
        6. It restores its execution context.
        7. It resumes its execution.
    c. When the OS receives any other I/O interrupt:
        1. It executes the I/O operation.
        2. It switches the thread that was waiting for that I/O operation, into the ready state.
        3. It resumes the execution of the current programs.
- The **Thread Control Block (TCB)** is a data structure in the operating system kernel which contains thread-specific information needed to manage it. It contains the following information:
    1. Tid (thread id)
    2. State (as either running, ready, waiting)
    3. Registers (including eip and esp)
    4. User (forthcoming lecture on user space)
    5. Pointer to a Process Control Block of the process that the thread lives on
- The OS maintains a collection of state queues with the TCBs of all the threads. There is one queue for the threads in the ready state and multiple queues for the threads in the waiting state (one queue for each type of I/O requests).
- **Synchronization:**
- **Process synchronization** is the task of coordinating the execution of processes in a way that no two processes can have access to the same shared data and resources.
- It is specially needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or data at the same time.
- One main error of synchronization is **race condition**. Race condition occurs when a system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.

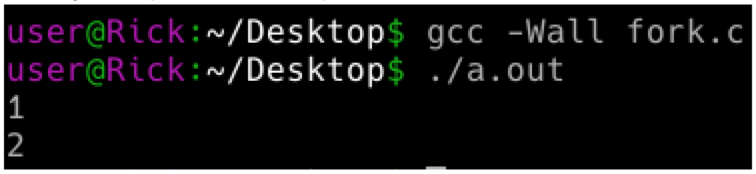
- A race condition is a condition of a program where its behavior depends on relative timing or interleaving of multiple threads or processes. One or more possible outcomes may be undesirable, resulting in a bug. We refer to this kind of behavior as **nondeterministic**.
- Race condition is very hard to catch and fix.
- Example of a race condition:



```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{

    // make two process which run same
    // program after this instruction
    int pid = fork();

    // Parent process
    if (pid > 0){
        printf("1\n");
    }

    // Child process
    else{
        printf("2\n");
    }
    return 0;
}
```

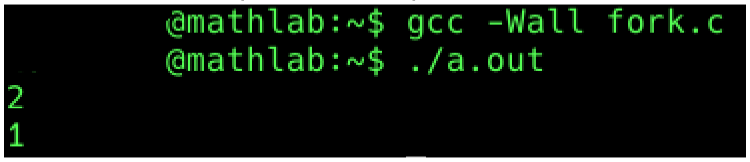
In the code above, we are using the fork() function to create a child process. Then, the parent process will print 1 while the child process will print 2.

On my computer, the output is 12.



```
user@Rick:~/Desktop$ gcc -Wall fork.c
user@Rick:~/Desktop$ ./a.out
1
2
```

On the UTSC computer, the output is 21.



```
        @mathlab:~$ gcc -Wall fork.c
        @mathlab:~$ ./a.out
2
1
```

In this example, the order doesn't really matter, but you can imagine in cases when order does matter, if the program executes in the wrong order, it can be disastrous.

Another issue with race condition is that the order executed depends on the computer the program is running on. Therefore, it can be hard to detect and fix.

- We want to use **mutual exclusion** to synchronize access to shared resources.
- Code that uses mutual exclusion to synchronize its execution is called a **critical section**.
- Only one thread at a time can execute in the critical section.
- All other threads are forced to wait on entry. The entry to the critical section is handled by the wait() function, and it is represented as P().
- When a thread leaves a critical section, another can enter. The exit from a critical section is controlled by the signal() function, represented as V().
- In the critical section, only a single process can be executed. Other processes, waiting to execute their critical section, need to wait until the current process completes its execution.
- Requirements for Critical Section:
    1. **Mutual Exclusion:** It states that if one thread is in the critical section, then no other is. Mutual exclusion ensures safety property (nothing bad happens).
    2. **Progress:** If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section. A thread in the critical section will eventually leave it.
    3. **Bound Waiting:** If some thread T is waiting on the critical section, then T will eventually enter the critical section. Progress and bounded waiting ensures the liveness property (something good happens).
    4. **Performance:** The overhead of entering and exiting the critical section is small with respect to the work being done within it.
- There are 3 main ways to achieve synchronization:
    1. **Lock/Mutex:**
        - This is a busy waiting solution which can be used for more than two processes.
        - In this mechanism, a variable or flag is used. The flag can have 2 values, 0 or 1. If the flag has a value of 0, that means the critical section is vacant. If the flag has a value of 1, that means the critical section is occupied.
        - A process which wants to get into the critical section first checks the value of the flag. If it is 0 then it sets the value to 1 and enters into the critical section, otherwise it waits.
        - The lock supports three operations:
          init() creates an unlocked mutex.
          acquire() waits until the mutex is unlocked, then locks it to enter the C.S.
          release() unlocks the mutex to leave the C.S, waking up anyone waiting for it.
    2. **Condition Variable:**
        - Condition variables are used to wait for a particular condition to become true.
        - A condition variable supports three operations:
          cond_wait(cond, lock) unlock the lock and sleep until cond is signaled then re-acquire lock before resuming execution.
          cond_signal(cond) signal the condition cond by waking up the next thread.
          cond_broadcast(cond) signal the condition cond by waking up all threads

3. **Semaphore:**
   - A semaphore is simply a variable that is non-negative and shared between threads. It is used to provide mutual exclusion.
   - A semaphore supports two operations:
     P()/decrement() blocks until semaphore is open.
     V()/increment() allows another thread to enter.
   - A semaphore safety property is that the semaphore value is always greater than or equal to 0.
   - Associated with each semaphore is a queue of waiting threads.

     When P() is called by a thread:
     - If semaphore is open, the thread continues.
     - If semaphore is closed, the thread blocks on queue.

     Then V() opens the semaphore:
     - If a thread is waiting on the queue, the thread is unblocked.
     - If no threads are waiting on the queue, the signal is remembered for the next thread.

- Semaphores may cause deadlock. **Deadlock** occurs when one thread tries to access a resource that a second holds, and vice-versa.

Textbook Notes:

- **Concurrency - An Introduction:**
- Concurrency - An Introduction:
- A **thread** is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, and a thread ID.
- A **multi-threaded program** has more than one point of execution. Another way to think of this is that each thread is very much like a separate process, except for one difference: they share the same address space and thus can access the same data.
- The state of a single thread is thus very similar to that of a process. It has a **program counter (PC)** that tracks where the program is fetching instructions from. Each thread has its own private set of **registers** it uses for computation; thus, if there are two threads, T1 and T2, that are running on a single processor, when switching from running T1 to running T2, a **context switch** must take place.
- One other major difference between threads and processes concerns the stack. In a single-threaded process, there is a single stack, usually residing at the bottom of the address space. However, in a multi-threaded process, each thread runs independently and of course may call into various routines to do whatever work it is doing. That means, there will be a stack per thread. Any stack-allocated variables, parameters, return values, and other things that we put on the stack will be placed in what is sometimes called **thread-local storage**, the stack of the relevant thread.

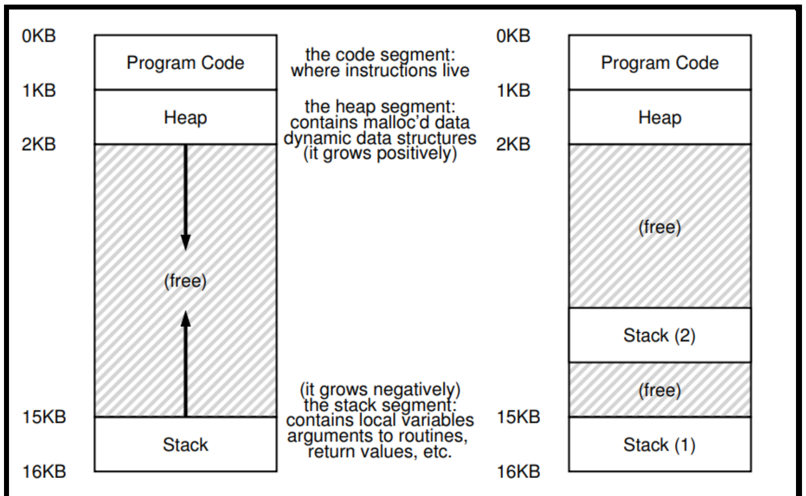

0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack
the code segment: where instructions live
the heap segment: contains malloc'd data dynamic data structures (it grows positively)
(it grows negatively) the stack segment: contains local variables arguments to routines, return values, etc.
0KB
1KB
2KB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

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

- Why Use Threads?:
- There are at least two major reasons you should use threads:
    1. **Parallelism.** Imagine you are writing a program that adds two large arrays together. If you are running on just a single processor, the task is straightforward: just perform each operation and be done. However, if you are executing the program on a system with multiple processors, you have the potential of speeding up this process considerably by using the processors to each perform a portion of the work. The task of transforming your standard single-threaded program into a program that does this sort of work on multiple CPUs is called **parallelization**, and using a thread per CPU to do this work is a natural and typical way to make programs run faster on modern hardware.
    2. To avoid blocking program progress due to slow I/O. Imagine that you are writing a program that performs different types of I/O. Instead of waiting, your program may wish to do something else, including utilizing the CPU to perform computation, or even issuing further I/O requests. Using threads is a natural way to avoid getting stuck; while one thread in your program **waits** (is blocked waiting for I/O), the CPU scheduler can switch to other threads, which are ready to run and do something useful. Threading enables overlap of I/O with other activities within a single program, much like multiprogramming did for processes across programs. As a result, many modern server-based applications (web servers, database management systems, etc) make use of threads in their implementations.
- An Example - Thread Creation:
- One issue with concurrency and multi-threading is **race condition**. Consider the following example: We have a function that creates 2 threads, T1 and T2. T1 will print "A" and T2 will print "B" and we want the order of the letters printed to be AB. Unfortunately, if there's no control process or scheduler, then we might get different outputs depending on different computers.
- Why It Gets Worse - Shared Data:
- Now, consider the previous example, but this time, instead of T1 and T2 printing A and B, respectively, they will be updating shared data. Suppose that we have a variable called num, initialized to be 0, and that T1 performs some mathematical operations and updates num and T2 uses the new value of num for another function. If T2 runs before T1 updates num, then the end result will be incorrect.
- The Heart Of The Problem - Uncontrolled Scheduling:
- What we have demonstrated here is called **race condition**. The results depend on the timing execution of the code. With some bad luck, such as context switches that occur at untimely points in the execution, we will get the wrong result. In fact, we may get a different result each time; thus, instead of a nice deterministic computation, which we are used to from computers, we call this result indeterminate, where it is not known what the output will be and it is indeed likely to be different across runs. Because multiple threads executing this code can result in a race condition, we call this code a **critical section**. A critical section is a piece of code that accesses a shared variable and must not be concurrently executed by more than one thread. What we really want for this code is what we call **mutual exclusion**. This property guarantees that if one thread is executing within the critical section, the others will be prevented from doing so.

- <u>The Wish For Atomicity:</u>
- **Atomic operations** are one of the most powerful underlying techniques in building computer systems, from the computer architecture, to concurrent code, to file systems, database management systems, and even distributed systems.
- The idea behind making a series of actions atomic is simply expressed with the phrase "all or nothing"; it should either appear as if all of the actions you wish to group together occurred, or that none of them occurred, with no in-between state visible. Sometimes, the grouping of many actions into a single atomic action is called a **transaction**, an idea developed in great detail in the world of databases and transaction processing.
- **Locks:**
- <u>Locks - The Basic Idea:</u>
- A **lock** is just a variable, and thus to use one, you must declare a lock variable of some kind. This lock variable holds the state of the lock at any instant in time. It is either available and thus no thread holds the lock, or acquired and thus exactly one thread holds the lock and presumably is in a critical section. We could store other information in the data type as well, such as which thread holds the lock, or a queue for ordering lock acquisition, but information like that is hidden from the user of the lock.
- Furthermore, we will have 2 functions, lock() and unlock(). Calling lock() tries to acquire the lock. If the lock is free, then the thread will acquire the lock and enter the critical section. This thread is sometimes said to be the owner of the lock. If another thread then calls lock() on that same lock variable, it will not return while the lock is held by another thread. In this way, other threads are prevented from entering the critical section while the first thread that holds the lock is in there.
- Once the owner of the lock calls unlock(), the lock is now free again. If no other threads are waiting for the lock, the state of the lock is simply changed to free. If there are waiting threads, one of them will notice this change of the lock's state, acquire the lock, and enter the critical section.
- Locks provide some minimal amount of control over scheduling to programmers. In general, we view threads as entities created by the programmer but scheduled by the OS, in any fashion that the OS chooses. Locks yield some of that control back to the programmer; by putting a lock around a section of code, the programmer can guarantee that no more than a single thread can ever be active within that code. Thus locks help transform the chaos that is traditional OS scheduling into a more controlled activity.
- <u>Pthread Locks:</u>
- The name that the POSIX library uses for a lock is a **mutex**, as it is used to provide mutual exclusion between threads.
- Here is the POSIX library lock code:
  ```
  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
  Pthread_mutex_lock(&lock); // wrapper; exits on failure
  balance = balance + 1;
  Pthread_mutex_unlock(&lock);
  ```
- You might also notice here that the POSIX version passes a variable to lock and unlock, as we may be using different locks to protect different variables. Doing so can increase concurrency. Instead of one big lock that is used any time any critical section is accessed, one will often protect different data and data structures with different locks, thus allowing more threads to be in locked code at once.

- Evaluating Locks:
- There are 3 basic criterias for locks to be useful:
    1. Whether the lock provides **mutual exclusion**.
    2. Is there **fairness**? Does each thread contending for the lock get a fair shot at acquiring it once it is free? Another way to look at this is by examining the more extreme case: does any thread contending for the lock starve while doing so, thus never obtaining it?
    3. The final criterion is **performance**, specifically the time overheads added by using the lock. There are a few different cases that are worth considering here. One is the case of no contention; when a single thread is running and grabs and releases the lock, what is the overhead of doing so? Another is the case where multiple threads are contending for the lock on a single CPU; in this case, are there performance concerns? Finally, how does the lock perform when there are multiple CPUs involved, and threads on each contending for the lock?
- Controlling Interrupts:
- One of the earliest solutions used to provide mutual exclusion was to disable interrupts for critical sections. This solution was invented for single-processor systems.
- By turning off interrupts before entering a critical section, we ensure that the code inside the critical section will not be interrupted, and thus will execute as if it were atomic. When we are finished, we re-enable interrupts and thus the program proceeds as usual.
- The main positive of this approach is its simplicity. You certainly don't have to scratch your head too hard to figure out why this works. Without interruption, a thread can be sure that the code it executes will execute and that no other thread will interfere with it.
- The negatives, unfortunately, are many:
    1. This approach requires us to allow any calling thread to perform a privileged operation (turning interrupts on and off), and thus trust that this facility is not abused. As you already know, any time we are required to trust an arbitrary program, we are probably in trouble. Here, the trouble manifests in numerous ways: a greedy program could call lock() at the beginning of its execution and thus monopolize the processor; worse, an errant or malicious program could call lock() and go into an endless loop. In this latter case, the OS never regains control of the system, and there is only one recourse: restart the system. Using interrupt disabling as a general purpose synchronization solution requires too much trust in applications.
    2. This approach does not work on multiprocessors. If multiple threads are running on different CPUs, and each try to enter the same critical section, it does not matter whether interrupts are disabled; threads will be able to run on other processors, and thus could enter the critical section. As multiprocessors are now commonplace, our general solution will have to do better than this.
    3. Turning off interrupts for extended periods of time can lead to interrupts becoming lost, which can lead to serious systems problems. Imagine, for example, if the CPU missed the fact that a disk device has finished a read request. How will the OS know to wake the process waiting for said read?
    4. This approach can be inefficient. Compared to normal instruction execution, code that masks or unmasks interrupts tends to be executed slowly by modern CPUs.

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

```
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;
}
```

- In this first attempt, the idea is quite simple: use a simple variable to indicate whether some thread has possession of a lock. The first thread that enters the critical section will call lock(), which tests whether the flag is equal to 1 and then sets the flag to 1 to indicate that the thread now holds the lock. When finished with the critical section, the thread calls unlock() and clears the flag, thus indicating that the lock is no longer held. If another thread happens to call lock() while that first thread is in the critical section, it will simply spin-wait in the while loop for that thread to call unlock() and clear the flag. Once that first thread does so, the waiting thread will fall out of the while loop, set the flag to 1 for itself, and proceed into the critical section.
- There are 2 problems with this design:
    1. With untimely interrupts, we can have 2 threads both set the variable to 1.
    2. This design has bad performance. While a thread waits to acquire a lock that is already held it endlessly checks the value of the flag, a technique known as **spin-waiting**. Spin-waiting wastes time waiting for another thread to release a lock. The waste is exceptionally high on a uniprocessor, where the thread that the waiter is waiting for cannot even run.

- <u>Building Working Spin Locks with Test-And-Set:</u>
- Because disabling interrupts does not work on multiple processors, and because simple approaches using loads and stores don't work, system designers started to invent hardware support for locking.
- The simplest bit of hardware support to understand is known as a **test-and-set** (or **atomic exchange**) instruction.
- Here is the code:

```
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
}
```

```
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;
}
```

- What the test-and-set instruction does is as follows. It returns the old value pointed to by the old ptr, and simultaneously updates said value to new. The key is that this sequence of operations is performed atomically. The reason it is called "test and set" is that it enables you to "test" the old value while simultaneously setting the memory location to a new value. As it turns out, this slightly more powerful instruction is enough to build a simple spin lock.
- Let's make sure we understand why this lock works. Imagine first the case where a thread calls lock() and no other thread currently holds the lock; thus, flag should be 0. When the thread calls TestAndSet(flag, 1), the routine will return the old value of flag, which is 0; thus, the calling thread, which is testing the value of flag, will not get caught spinning in the while loop and will acquire the lock. The thread will also atomically set the value to 1, thus indicating that the lock is now held. When the thread is finished with its critical section, it calls unlock() to set the flag back to zero.
- The second case we can imagine arises when one thread already has the lock held (i.e., flag is 1). In this case, this thread will call lock() and then call TestAndSet(flag, 1) as well. This time, TestAndSet() will return the old value at flag, which is 1 (because the lock is held), while simultaneously setting it to 1 again. As long as the lock is held by another thread, TestAndSet() will repeatedly return 1, and thus this thread will spin and spin until the lock is finally released. When the flag is finally set to 0 by some other thread, this thread will call TestAndSet() again, which will now return 0 while atomically setting the value to 1 and thus acquire the lock and enter the critical section.

- By making both the test and set a single atomic operation, we ensure that only one thread acquires the lock. And that's how to build a working mutual exclusion primitive.
- You may also now understand why this type of lock is usually referred to as a **spin lock**. It is the simplest type of lock to build, and simply spins, using CPU cycles, until the lock becomes available. To work correctly on a single processor, it requires a **preemptive scheduler** (one that will interrupt a thread via a timer, in order to run a different thread, from time to time). Without preemption, spin locks don't make much sense on a single CPU, as a thread spinning on a CPU will never relinquish it.
- Evaluating Spin Locks:
- Given our basic spin lock, we can now evaluate how effective it is along our previously described axes. The most important aspect of a lock is correctness: does it provide mutual exclusion? The answer here is yes: the spin lock only allows a single thread to enter the critical section at a time. Thus, we have a correct lock.
- The next axis is fairness. How fair is a spin lock to a waiting thread? Can you guarantee that a waiting thread will ever enter the critical section? The answer here, unfortunately, is bad news: spin locks don't provide any fairness guarantees. Indeed, a thread spinning may spin forever, under contention. Simple spin locks, as discussed thus far, are not fair and may lead to starvation.
- The final axis is performance. What are the costs of using a spin lock? To analyze this more carefully, we suggest thinking about a few different cases. In the first, imagine threads competing for the lock on a single processor; in the second, consider threads spread out across many CPUs. For spin locks, in the single CPU case, performance overheads can be quite painful; imagine the case where the thread holding the lock is preempted within a critical section. The scheduler might then run every other thread (imagine there are N − 1 others), each of which tries to acquire the lock. In this case, each of those threads will spin for the duration of a time slice before giving up the CPU, a waste of CPU cycles. However, on multiple CPUs, spin locks work reasonably well, if the number of threads roughly equals the number of CPUs. The thinking goes as follows: imagine Thread A on CPU 1 and Thread B on CPU 2, both contending for a lock. If Thread A (CPU 1) grabs the lock, and then Thread B tries to, B will spin (on CPU 2). However, presumably the critical section is short, and thus soon the lock becomes available, and is acquired by Thread B. Spinning to wait for a lock held on another processor doesn't waste many cycles in this case, and thus can be effective.
- Compare-And-Swap:
- Another hardware primitive that some systems provide is known as the **compare-and-swap** instruction or **compare-and-exchange**.
- Here is the code:

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

- The basic idea is for compare-and-swap to test whether the value at the address specified by ptr is equal to expected. If so, update the memory location pointed to by ptr with the new value. If not, do nothing. In either case, return the original value at that memory location, thus allowing the code calling compare-and-swap to know whether it succeeded or not.

- Load-Linked and Store-Conditional:
- Some platforms provide a pair of instructions that work in concert to help build critical sections. On the MIPS architecture, for example, the **load-linked** and **store-conditional** instructions can be used in tandem to build locks and other concurrent structures.
- Here is the code:

```
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no update to *ptr since LoadLinked to this address) {
        *ptr = value;
        return 1; // success!
    } else {
        return 0; // failed to update
    }
}
```

- The load-linked operates much like a typical load instruction, and simply fetches a value from memory and places it in a register. The key difference comes with the store-conditional, which only succeeds (and updates the value stored at the address just load-linked from) if no intervening store to the address has taken place. In the case of success, the storeconditional returns 1 and updates the value at ptr to value; if it fails, the value at ptr is not updated and 0 is returned.
- Fetch-And-Add:
- One final hardware primitive is the **fetch-and-add** instruction, which atomically increments a value while returning the old value at a particular address.
- Here is the code:

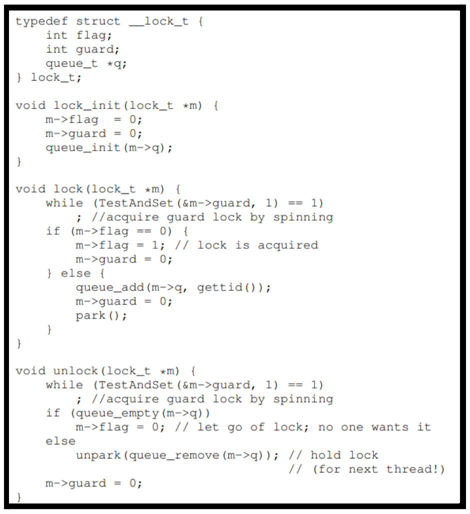
```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

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;
}
```

- Instead of a single value, this solution uses a ticket and turn variable in combination to build a lock. The basic operation is pretty simple: when a thread wishes to acquire a lock, it first does an atomic fetch-and-add on the ticket value; that value is now considered this thread's "turn" (myturn). The globally shared lock->turn is then used to determine which thread's turn it is; when (myturn == turn) for a given thread, it is that thread's turn to enter the critical section. Unlock is accomplished simply by incrementing the turn such that the next waiting thread, if there is one, can now enter the critical section.
- Note one important difference with this solution versus our previous attempts: it ensures progress for all threads. Once a thread is assigned its ticket value, it will be scheduled at some point in the future. In our previous attempts, no such guarantee existed; a thread

spinning on test-and-set could spin forever even as other threads acquire and release the lock.

- <u>A Simple Approach - Just Yield:</u>
- Hardware support got us pretty far: working locks, and even fairness in lock acquisition. However, we still have a problem: what to do when a context switch occurs in a critical section, and threads start to spin endlessly, waiting for the interrupted (lock-holding) thread to be run again?
- Our first try is a simple and friendly approach: when you are going to spin, instead give up the CPU to another thread.
- In this approach, we assume an operating system primitive yield() which a thread can call when it wants to give up the CPU and let another thread run. A thread can be in one of three states (running, ready, or blocked). Yield is simply a system call that moves the caller from the running state to the ready state, and thus promotes another thread to running. Thus, the yielding thread essentially deschedules itself.
- Think about the example with two threads on one CPU; in this case, our yield-based approach works quite well. If a thread happens to call lock() and find a lock held, it will simply yield the CPU, and thus the other thread will run and finish its critical section. In this simple case, the yielding approach works well.
- Let us now consider the case where there are many threads (say 100) contending for a lock repeatedly. In this case, if one thread acquires the lock and is preempted before releasing it, the other 99 will each call lock(), find the lock held, and yield the CPU. Assuming some kind of round-robin scheduler, each of the 99 will execute this run-and-yield pattern before the thread holding the lock gets to run again. While better than our spinning approach, this approach is still costly; the cost of a context switch can be substantial, and there is thus plenty of waste. Worse, we have not tackled the starvation problem at all. A thread may get caught in an endless yield loop while other threads repeatedly enter and exit the critical section. We clearly will need an approach that addresses this problem directly.
- One good reason to avoid spin locks is performance. If a thread is interrupted while holding a lock, other threads that use spin locks will spend a large amount of CPU time just waiting for the lock to become available. However, it turns out there is another interesting reason to avoid spin locks on some systems: **priority inversion**. Priority inversion is a scenario in scheduling in which a high priority task is indirectly preempted by a lower priority task effectively inverting the relative priorities of the two tasks.
- Let's assume there are two threads in a system. Thread 2 (T2) has a high scheduling priority, and Thread 1 (T1) has lower priority. In this example, let's assume that the CPU scheduler will always run T2 over T1, if indeed both are runnable; T1 only runs when T2 is not able to do so (e.g., when T2 is blocked on I/O). Now, the problem. Assume T2 is blocked for some reason. So T1 runs, grabs a spin lock, and enters a critical section. T2 now becomes unblocked (perhaps because an I/O completed), and the CPU scheduler immediately schedules it (thus descheduling T1). T2 now tries to acquire the lock, and because it can't (T1 holds the lock), it just keeps spinning. Because the lock is a spin lock, T2 spins forever, and the system is hung.
- Just avoiding the use of spin locks, unfortunately, does not avoid the problem of inversion. Imagine three threads, T1, T2, and T3, with T3 at the highest priority, and T1 the lowest. Imagine now that T1 grabs a lock. T3 then starts, and because it is higher priority than T1, runs immediately (preempting T1). T3 tries to acquire the lock that T1 holds, but gets stuck waiting, because T1 still holds it. If T2 starts to run, it will have higher priority than T1, and thus it will run. T3, which is higher priority than T2, is stuck waiting for T1, which may never run now that T2 is running.

- You can address the priority inversion problem in a number of ways. In the specific case where spin locks cause the problem, you can avoid using spin locks. More generally, a higher-priority thread waiting for a lower-priority thread can temporarily boost the lower thread's priority, thus enabling it to run and overcoming the inversion, a technique known as priority inheritance. A last solution is simplest: ensure all threads have the same priority.
- <u>Using Queues - Sleeping Instead Of Spinning:</u>
- The real problem with our previous approaches is that they leave too much to chance. The scheduler determines which thread runs next; if the scheduler makes a bad choice, a thread runs that must either spin waiting for the lock, or yield the CPU immediately. Either way, there is potential for waste and no prevention of starvation.
- For simplicity, we will use the support provided by Solaris, in terms of two calls: park() to put a calling thread to sleep, and unpark(threadID) to wake a particular thread as designated by threadID. These two routines can be used in tandem to build a lock that puts a caller to sleep if it tries to acquire a held lock and wakes it when the lock is free.
- We do a couple of interesting things in this example. First, we combine the old test-and-set idea with an explicit queue of lock waiters to make a more efficient lock. Second, we use a queue to help control who gets the lock next and thus avoid starvation.
- Here is the code:



```
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;
}
```

- You might notice how the guard is used basically as a spin-lock around the flag and queue manipulations the lock is using. This approach thus doesn't avoid spin-waiting entirely. A thread might be interrupted while acquiring or releasing the lock, and thus cause other threads to spin-wait for this one to run again. However, the time spent spinning is quite limited (just a few instructions inside the lock and unlock code, instead of the user-defined critical section), and thus this approach may be reasonable.
- You might also observe that in lock(), when a thread can not acquire the lock, we are careful to add ourselves to a queue, set guard to 0, and yield the CPU. A question for the reader: What would happen if the release of the guard lock came after the park(), and not before? Hint: something bad.
- You might further detect that the flag does not get set back to 0 when another thread gets woken up. This is not an error, but rather a necessity. When a thread is woken up, it will be as if it is returning from park(); however, it does not hold the guard at that point in the code and thus cannot even try to set the flag to 1. Thus, we just pass the lock directly from the thread releasing the lock to the next thread acquiring it.

- Finally, you might notice the perceived race condition in the solution, just before the call to park(). With just the wrong timing, a thread will be about to park, assuming that it should sleep until the lock is no longer held. A switch at that time to another thread (say, a thread holding the lock) could lead to trouble, for example, if that thread then released the lock. The subsequent park by the first thread would then sleep forever, a problem sometimes called the **wakeup/waiting race**.
- Solaris solves this problem by adding a third system call: setpark(). By calling this routine, a thread can indicate it is about to park. If it then happens to be interrupted and another thread calls unpark before park is actually called, the subsequent park returns immediately instead of sleeping.
- **Semaphores:**
- Semaphores - A Definition:
- A **semaphore** is an object with an integer value that we can manipulate with two routines; in the POSIX standard, these routines are sem wait() and sem post().
- Because the initial value of the semaphore determines its behavior, before calling any other routine to interact with the semaphore, we must first initialize it to some value.
- Here is the code for initializing a semaphore:

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);
```

- In the code, we declare a semaphore s and initialize it to the value 1 by passing 1 in as the third argument. The second argument to sem_init() will be set to 0 in all of the examples we'll see. This indicates that the semaphore is shared between threads in the same process.
- After a semaphore is initialized, we can call one of two functions to interact with it, sem wait() or sem post().
- Here is the implementation for sem_wait() and sem_post():

```
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
}
```

- **sem_wait(sem_t *sem)** decrements (locks) the semaphore pointed to by sem. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement, or a signal handler interrupts the call.
- **sem_post(sem_t *sem)** increments (unlocks) the semaphore pointed to by sem. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a sem_wait() call will be woken up and proceed to lock the semaphore.

- <u>Binary Semaphores (Locks):</u>
- Our first use will be one with which we are already familiar: using a semaphore as a lock.
- Here is the code:

```
sem_t m;
sem_init(&m, 0, X); // initialize to X; what should X be?

sem_wait(&m);
// critical section here
sem_post(&m);
```

- Looking back at the definition of the sem wait() and sem post() routines above, we can see that the initial value should be 1.

  To make this clear, let's imagine a scenario with two threads. The first thread (Thread 0) calls sem wait(); it will first decrement the value of the semaphore, changing it to 0. Then, it will wait only if the value is not greater than or equal to 0. Because the value is 0, sem wait() will simply return and the calling thread will continue; Thread 0 is now free to enter the critical section. If no other thread tries to acquire the lock while Thread 0 is inside the critical section, when it calls sem post(), it will simply restore the value of the semaphore to 1 and not wake a waiting thread, because there are none.
  Here is a trace of the above example:

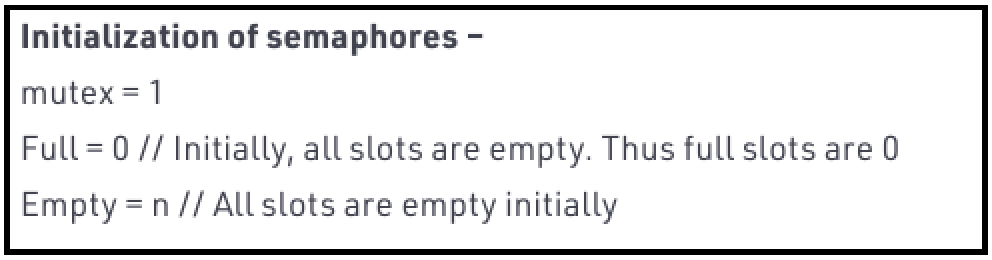
Value of Semaphore	Thread 0	Thread 1
1		
1	call sem_wait()	
0	sem_wait() returns	
0	(crit sect)	
0	call sem_post()	
1	sem_post() returns	

  A more interesting case arises when Thread 0 holds the lock (it has called sem wait() but not yet called sem post()), and another thread (Thread 1) tries to enter the critical section by calling sem wait(). In this case, Thread 1 will decrement the value of the semaphore to -1, and thus wait (putting itself to sleep and relinquishing the processor). When Thread 0 runs again, it will eventually call sem post(), incrementing the value of the semaphore back to zero, and then wake the waiting thread (Thread 1), which will then be able to acquire the lock for itself. When Thread 1 finishes, it will again increment the value of the semaphore, restoring it to 1 again.
  Here is a trace of the above example:

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

- Thus we are able to use semaphores as locks. Because locks only have two states (held and not held), we sometimes call a semaphore used as a lock a **binary semaphore**.

- Semaphores For Ordering:
- Semaphores are also useful to order events in a concurrent program.
- For example, a thread may wish to wait for a list to become non-empty, so it can delete an element from it. In this pattern of usage, we often find one thread waiting for something to happen, and another thread making that something happen and then signaling that it has happened, thus waking the waiting thread.
- The Producer/Consumer (Bounded Buffer) Problem:
- The next problem we will confront in this chapter is known as the **producer/consumer problem**, or sometimes as the **bounded buffer problem**.
- Problem Statement: We have a buffer of fixed size. A producer can produce an item and can place it in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time the consumer should not consume any item. In this problem, the buffer is the critical section.
- To solve this problem, we need two counting semaphores – Full and Empty. "Full" keeps track of the number of items in the buffer at any given time and "Empty" keeps track of the number of unoccupied slots.

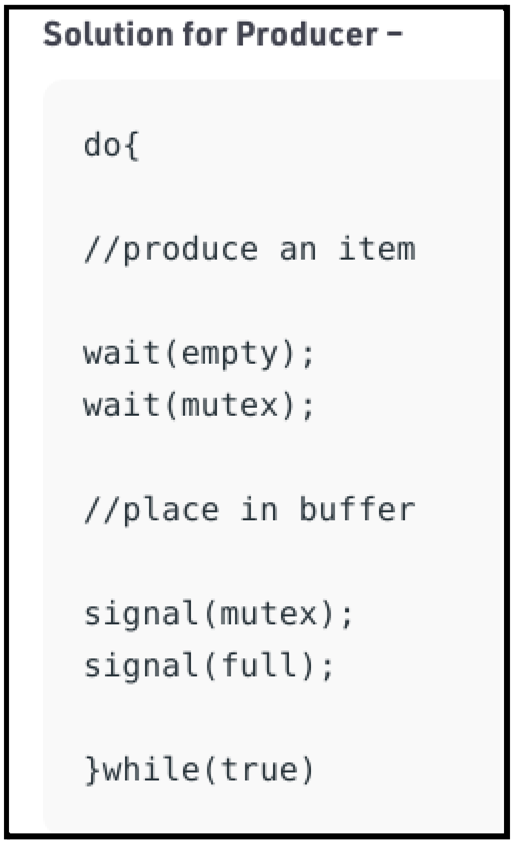


Initialization of semaphores –

mutex = 1

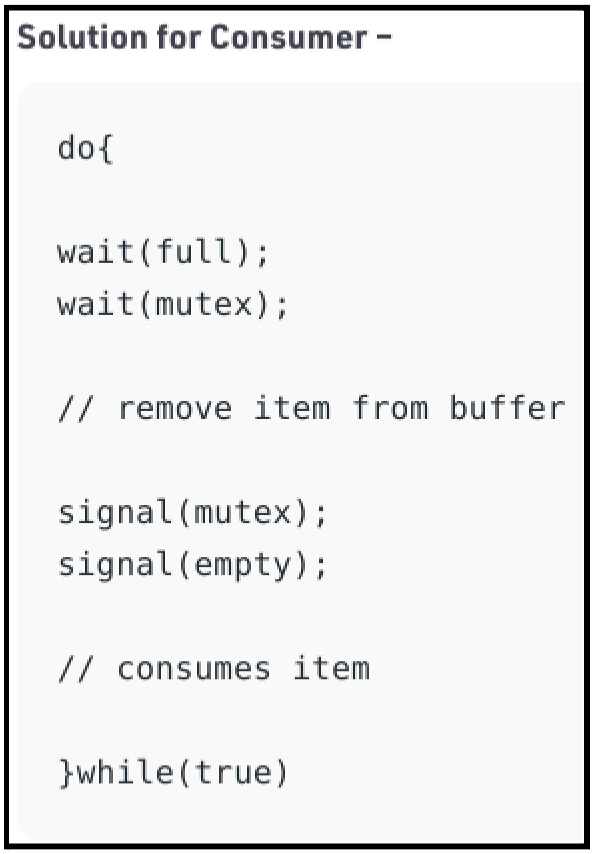
Full = 0 // Initially, all slots are empty. Thus full slots are 0

Empty = n // All slots are empty initially



Solution for Producer –

```
do{

//produce an item

wait(empty);
wait(mutex);

//place in buffer

signal(mutex);
signal(full);

}while(true)
```



Solution for Consumer –

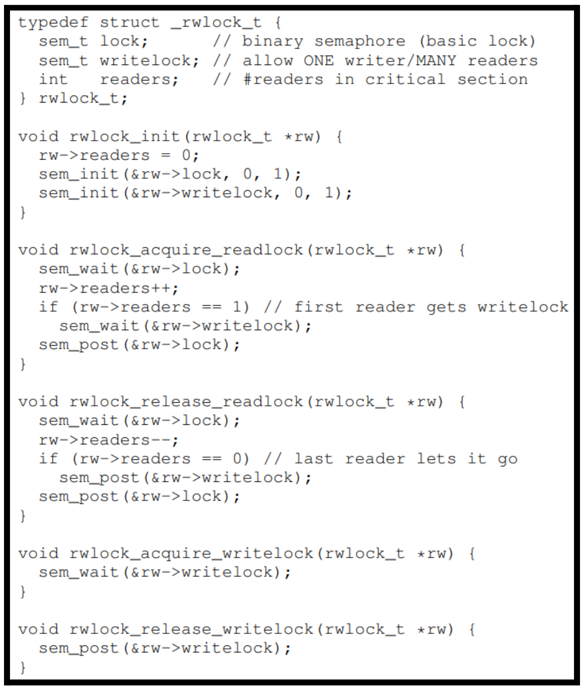
```
do{

wait(full);
wait(mutex);

// remove item from buffer

signal(mutex);
signal(empty);

// consumes item

}while(true)
```

When a producer produces an item then the value of “empty” is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of “full” is increased by 1. The value of mutex is also increased by 1 because the task of producer has been completed and consumer can access the buffer.

As the consumer is removing an item from the buffer, therefore the value of “full” is reduced by 1 and the value of mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of “empty” by 1. The value of mutex is also increased so that producer can access the buffer now.

I.e. Semaphores solve the problem of lost wakeup calls. In the solution below, we use two semaphores, Full and Empty, to solve the problem. Full is the number of items already in the buffer and available to be read, while Empty is the number of available spaces in the buffer where items could be written. Full is incremented and Empty decremented when a new item is put into the buffer. If the producer tries to decrement Empty when its value is zero, the producer is put to sleep. The next time an item is consumed, Empty is incremented and the producer wakes up. The consumer works analogously.

- <u>Reader-Writer Locks:</u>
- Another classic problem stems from the desire for a more flexible locking primitive that admits that different data structure accesses might require different kinds of locking.
- For example, imagine a number of concurrent list operations, including inserts and simple lookups. While inserts change the state of the list, lookups simply read the data structure; as long as we can guarantee that no insert is on-going, we can allow many lookups to proceed concurrently. The special type of lock we will now develop to support this type of operation is known as a **reader-writer lock**.
- Here is the code:



```
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);
}
```

- The code is pretty simple. If some thread wants to update the data structure in question, it should call the new pair of synchronization operations: rwlock acquire writelock(), to acquire a write lock, and rwlock release writelock(), to release it. Internally, these simply use the writelock semaphore to ensure that only a single writer can acquire the lock and thus enter the critical section to update the data structure in question.
- More interesting is the pair of routines to acquire and release read locks. When acquiring a read lock, the reader first acquires lock and then increments the readers variable to track how many readers are currently inside the data structure. The important step then taken within rwlock to acquire readlock() occurs when the first reader acquires the lock; in that case, the reader also acquires the write lock by calling sem wait() on the writelock semaphore, and then releasing the lock by calling sem post().
- Thus, once a reader has acquired a read lock, more readers will be allowed to acquire the read lock too; however, any thread that wishes to acquire the write lock will have to wait until all readers are finished; the last one to exit the critical section calls sem post() on "writelock" and thus enables a waiting writer to acquire the lock.
- This approach works as desired, but does have some negatives, especially when it comes to fairness. In particular, it would be relatively easy for readers to starve writers.
- Finally, it should be noted that reader-writer locks should be used with some caution. They often add more overhead and thus do not end up speeding up performance as compared to just using simple and fast locking primitives.
- Thread Throttling:
- One other simple use case for semaphores is this: How can a programmer prevent too many threads from doing something at once and bogging the system down? Answer: decide upon a threshold for too many, and then use a semaphore to limit the number of threads concurrently executing the piece of code in question. We call this approach **thread throttling**, and consider it a form of admission control.
- Let's consider a more specific example. Imagine that you create hundreds of threads to work on some problem in parallel. However, in a certain part of the code, each thread acquires a large amount of memory to perform part of the computation; let's call this part of the code the memory-intensive region. If all of the threads enter the memory-intensive region at the same time, the sum of all the memory allocation requests will exceed the amount of physical memory on the machine. As a result, the machine will start thrashing, and the entire computation will slow to a crawl.

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

- **Common Concurrency Problems:**
- Non-Deadlock Bugs:
    - **Atomicity-Violation Bugs:** The desired serializability among multiple memory accesses is violated.
      E.g.
      ```
      Thread 1::
      if (thd->proc_info) {
        fputs(thd->proc_info, ...);
      }

      Thread 2::
      thd->proc_info = NULL;
      ```
      In the example, two different threads access the field proc info in the structure thd. The first thread checks if the value is non-NULL and then prints its value; the second thread sets it to NULL. Clearly, if the first thread performs the check but then is interrupted before the call to fputs, the second thread could run in-between, thus setting the pointer to NULL; when the first thread resumes, it will crash, as a NULL pointer will be dereferenced by fputs.
    - **Order-Violation Bugs:** The desired order between two groups of memory accesses is flipped. The fix to this type of bug is generally to enforce ordering. As discussed previously, using condition variables is an easy and robust way to add this style of synchronization into modern code bases.
      E.g.
      ```
      Thread 1::
      void init() {
          mThread = PR_CreateThread(mMain, ...);
      }

      Thread 2::
      void mMain(...) {
           mState = mThread->State;
      }
      ```
      The code in Thread 2 seems to assume that the variable mThread has already been initialized and is not NULL. However, if Thread 2 runs immediately once created, the value of mThread will not be set when it is accessed within mMain() in Thread 2, and will likely crash with a NULL-pointer dereference. Note that we assume the value of mThread is initially NULL; if not, even stranger things could happen as arbitrary memory locations are accessed through the dereference in Thread 2.
- Deadlock Bugs:
- Beyond the concurrency bugs mentioned above, a classic problem that arises in many concurrent systems with complex locking protocols is known as **deadlock**.
- Deadlock occurs, for example, when a thread (Thread 1) is holding a lock (L1) and waiting for another one (L2). Unfortunately, the thread (Thread 2) that holds lock L2 is waiting for L1 to be released.

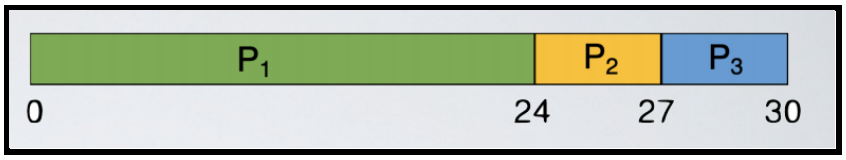
- One reason why deadlocks occur is that in large code bases, complex dependencies arise between components. Take the operating system, for example. The virtual memory system might need to access the file system in order to page in a block from disk. The file system might subsequently require a page of memory to read the block into and thus contact the virtual memory system. Thus, the design of locking strategies in large systems must be carefully done to avoid deadlock in the case of circular dependencies that may occur naturally in the code.
- Another reason is due to the nature of encapsulation. As software developers, we are taught to hide details of implementations and thus make software easier to build in a modular way. Unfortunately, such modularity does not mesh well with locking.
- Conditions for Deadlock - There are four conditions need to hold for a deadlock to occur:
    1. **Mutual exclusion:** Threads claim exclusive control of resources that they require.
    2. **Hold-and-wait:** Threads hold resources allocated to them while waiting for additional resources.
    3. **No preemption:** Resources cannot be forcibly removed from threads that are holding them.
    4. **Circular wait:** There exists a circular chain of threads such that each thread holds one or more resources that are being requested by the next thread in the chain.
- **Circular wait:** Probably the most practical prevention technique and certainly one that is frequently employed is to write your locking code such that you never induce a circular wait. The most straightforward way to do that is to provide a total ordering on lock acquisition. For example, if there are only two locks in the system (L1 and L2), you can prevent deadlock by always acquiring L1 before L2. Such strict ordering ensures that no cyclical wait arises; hence, no deadlock.

  Of course, in more complex systems, more than two locks will exist, and thus total lock ordering may be difficult to achieve. Thus, a partial ordering can be a useful way to structure lock acquisition so as to avoid deadlock.
- **Hold-and-wait:** The hold-and-wait requirement for deadlock can be avoided by acquiring all locks at once, atomically.
- **No Preemption:** Because we generally view locks as held until unlock is called, multiple lock acquisition often gets us into trouble because when waiting for one lock we are holding another. Many thread libraries provide a more flexible set of interfaces to help avoid this situation. Specifically, the routine pthread mutex trylock() either grabs the lock if it is available and returns success or returns an error code indicating the lock is held. In the latter case, you can try again later if you want to grab that lock.
- **Deadlock Avoidance via Scheduling:** Instead of deadlock prevention, in some scenarios deadlock avoidance is preferable. Avoidance requires some global knowledge of which locks various threads might grab during their execution, and subsequently schedules said threads in a way as to guarantee no deadlock can occur.
- **Detect and Recover:** One final general strategy is to allow deadlocks to occasionally occur, and then take some action once such a deadlock has been detected.

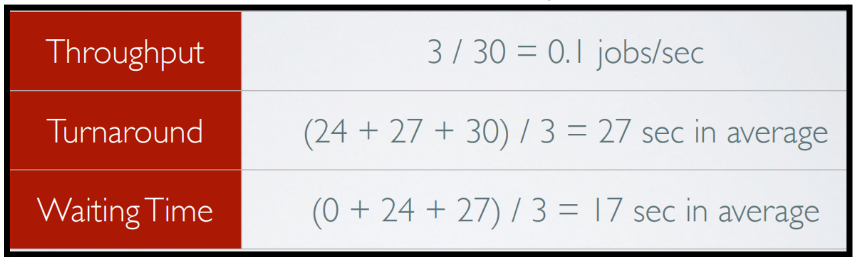
Lecture Notes:

- **The scheduling problem:**
- Suppose we have n threads ready to run and k CPUs, where k ≥ 1. Which jobs should we assign to which CPU(s) and for how long?
- Furthermore, we don't want **starvation** to occur.
- **Starvation** is when a thread is prevented from making progress because some other thread has the resource it requires (could be CPU or a lock).
- Starvation is usually a side effect of the scheduling algorithm.
  E.g. A high priority thread always prevents a low priority thread from running.
- Starvation can also be a side effect of synchronization.
- **Scheduling Criteria:**
- There are 3 main scheduling criterias:
  1. **Throughput:** This is the number of threads that complete per unit time.
     I.e. It is the number of jobs/time (Higher is better)
  2. **Turnaround time:** This is the time for each thread to complete.
     I.e. It is Time_finish – Time_start (Lower is better)
  3. **Response time:** This is the time from request to first response.
     I.e. It is the time between waiting to ready transition and ready to running transition.
     Time_response – Time_request (Lower is better)
- The above criterias are affected by secondary criteria. There are 2 secondary criterias:
  1. **CPU utilization:** This is the percent of time the CPU is doing productive work.
  2. **Waiting time:** This is the average time each thread waits in the ready queue.
- **How to balance criteria:**
- **Batch systems** (supercomputers) strive for job throughput and turnaround time.
- **Interactive systems** (personal computers) strive to minimize response time for interactive jobs. However, in practice, users prefer predictable response time over faster but highly variable response time. Often, personal computers are optimized for an average response time.
- **Two kinds of scheduling algorithms:**
  1. **Non-preemptive scheduling:**
  - This is good for batch systems.
  - Once the CPU has been allocated to a thread, it keeps the CPU until it terminates.
  - Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time.
  2. **Preemptive scheduling:**
  - This is good for interactive systems.
  - The CPU can be taken from a running thread and allocated to another.
  - Preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

- **FCFS - First Come First Serve:**
- A type of non-preemptive scheduling.
- Here, jobs are run in the order that they arrive and there are no interrupts.
- Consider this example:
  We have 3 threads, P1, P2 and P3 that all start at time 0 and go in that order.
  Suppose P1 takes 24 seconds, P2 takes 3 seconds and P3 takes 3 seconds.

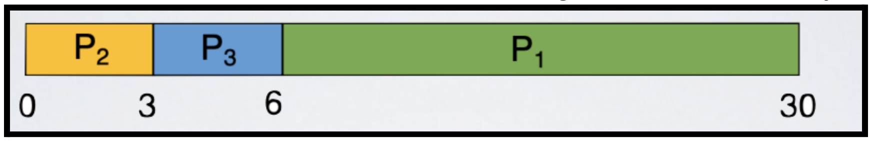


Here are the 3 main criterias for this example:

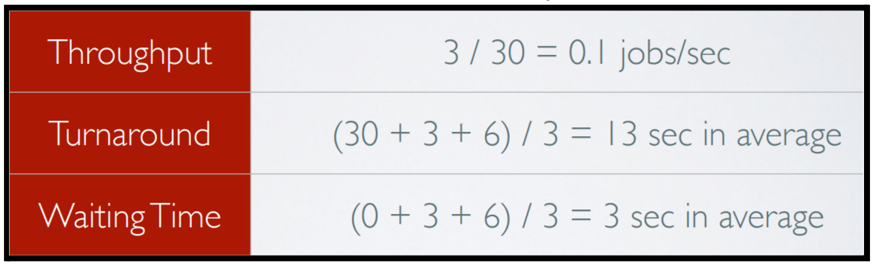


Throughput	3 / 30 = 0.1 jobs/sec
Turnaround	(24 + 27 + 30) / 3 = 27 sec in average
Waiting Time	(0 + 24 + 27) / 3 = 17 sec in average

- From this example, we can see a huge problem with this method. All other threads wait for the one big thread to release the CPU. This is known as the **Convoy Effect**.
- **SJF - Shortest-Job-First:**
- A type of non-preemptive scheduling.
- Here, jobs are run based on their processing time in order from least to greatest.
  I.e. Choose the thread with the shortest processing time.
- Consider this example:
  We have 3 threads, P1, P2 and P3 that all start at time 0.
  Suppose P1 takes 24 seconds, P2 takes 3 seconds and P3 takes 3 seconds.
  Since P2 takes the least amount of time, it goes first, followed by P3 and then P1.



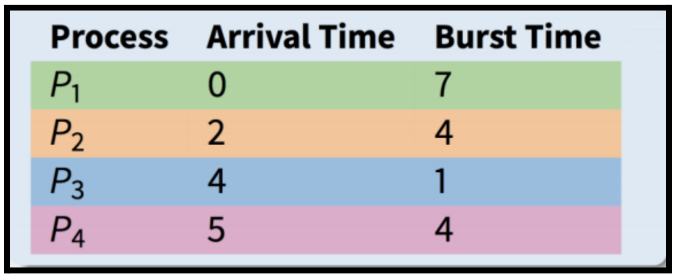
Here are the 3 main criterias for this example:



Throughput	3 / 30 = 0.1 jobs/sec
Turnaround	(30 + 3 + 6) / 3 = 13 sec in average
Waiting Time	(0 + 3 + 6) / 3 = 3 sec in average

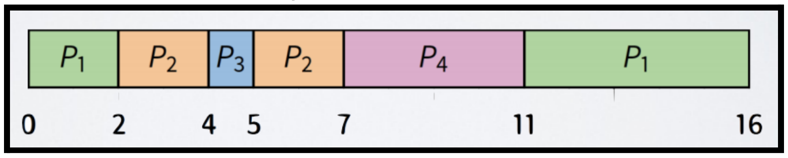
- The problem with this method is that we need to know processing time in advance.
- Easy to implement in batch systems where the required CPU time is known in advance.
- Impossible to implement in interactive systems where the required CPU time is not known.

- **SRTF - Shortest-Remaining-Time-First:**
- A type of preemptive scheduling.
- In this approach, if a new thread arrives with a CPU burst length less than the remaining time of the current executing thread, then preempt the current thread.
- Consider this example:
  We have 5 threads, P1, P2, P3, P4 and P5 with the following information



Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

This is the order each process runs and finishes.



P1 runs from time 0 to 2. However, at time 2, P2 arrives and since its burst time (4) is smaller than P1's remaining time (5), P2 now runs.

P2 runs from time 2 to 4. However, at time 4, P3 arrives and since its burst time (1) is smaller than P2's remaining time (2) and P1's remaining time (5), P3 now runs.

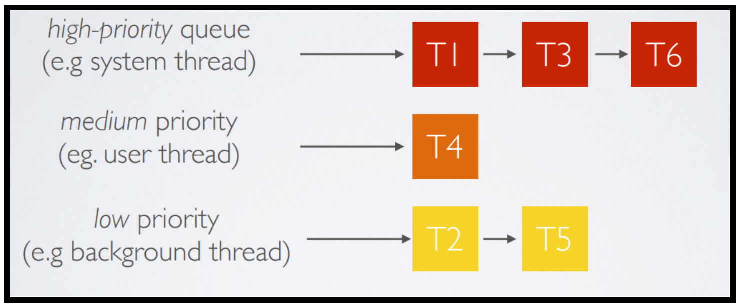
P3 runs from time 4 to 5 and finishes. At time 5, P4 arrives. Right now, P2 has the shortest time (2), compared to P4 (4) and P1 (5), so P2 runs.

P2 runs from time 5 to 7 and finishes. Now, we have 2 processes P1 and P4. P4 has a time of 4 while P1 has a time of 5, so P4 runs.

P4 runs from time 7 to 11 and finishes. Since P1 is the only process left, it runs from time 11 to 16 and finishes.

- The advantage of this method is that it optimizes waiting time.
- The problem with this method is that it can lead to starvation. In our example above, even though P1 was the first process created, it was the last to finish. Small processes can starve larger processes.
- **RR - Round Robin:**
- A type of preemptive scheduling.
- Each job is given a time slice called a **quantum**, and we preempt each job after the duration of its quantum, moving it to the back of the FIFO queue.
- The advantage of this method is that it has a fair allocation of CPU and a low waiting time (interactive).
- The problem with this method is that there is no priority between threads.
- Context switching is used to save states of preempted processes. Since context switches are frequent and need to be very fast, we want the quantum to be much larger than the context switch cost. The majority of bursts should be less than the quantum but you don't want the quantum to be so big that the system reverts to FCFS.
- Typical values for a quantum is between 1–100 ms.

- **MLQ - Multilevel Queue Scheduling:**
- A type of preemptive scheduling.
- Each thread is associated with a priority and the highest priority thread(s) are executed before lower priority thread(s). If multiple threads have the same priority, then do round-robin.
- E.g.



Here, we have 3 threads (T1, T3, T6) that are high priority, 1 thread (T4) that is medium priority and 2 threads (T2 and T5) that are low priority.

The high-priority threads will be executed first, and since there are multiple high-priority threads, round-robin will be used.

After all the high-priority threads have been executed, the medium-priority threads will be executed. Since there is only 1, there is no need to do round-robin.

After all the medium-priority threads have been executed, the low-priority threads will be executed, and since there are multiple low-priority threads, round-robin will be used.
- Some problems with this method:
    1. Starvation of low priority thread(s).
    2. Possible starvation of high priority thread(s).
    3. Priorities are arbitrarily decided. How do we decide on the priority?
- Example of starvation of high priority threads:
    - Suppose we have 3 threads, T1 (low priority), T2 (medium priority) and T3 (high priority) and a lock, L.
    - T1 starts, runs and acquires L.
    - T2 starts, preempts the CPU and runs.
    - T3 starts, preempts the CPU, runs but gets blocked while trying to acquire L.
    - T2 is elected to run as it is the highest priority thread to be ready to run.

    - As you can see, T3 (the high priority thread) is now starved.
    - A solution to this issue is **priority donation**.
- Example of priority donation:
    - Suppose we have 3 threads, T1 (low priority), T2 (medium priority) and T3 (high priority) and a lock, L.
    - T1 starts, runs and acquires L.
    - T2 starts, preempts the CPU and runs.
    - T3 starts, preempts the CPU, runs but gets blocked while trying to acquire L.
    - T3 gives its high priority to T1.
    - T1 (now high priority) runs, releases the lock and returns to low priority immediately after.
    - T3 (now unblocked) preempts the CPU and runs.

- To prevent starvation of low priority threads, change the priority over time by either increase the priority as a function of waiting time or decrease the priority as a function of CPU consumption.
- To decide on the priority, observe and keep track of the thread CPU usage.
- **MLFQ - Multilevel Feedback Queue Scheduling:**
- A type of preemptive scheduling.
- This is the same as MLQ but it changes the priority of the process based on observations.
- This is a Turing-award winner algorithm.
- Observations:

Rule 1	If Priority(A) > Priority(B), A runs
Rule 2	If Priority(A) = Priority(B), A & B run in round-robin fashion using the time slice (quantum length) of the given queue
Rule 3	When a job enters the system, it is placed at the highest priority (the topmost queue)
Rule 4	Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue)
Rule 5	After some time period S, move all the jobs in the system to the topmost queue

Textbook Notes:

- **Mechanism - Limited Direct Execution:**
- Introduction:
- In order to virtualize the CPU, the operating system needs to somehow share the physical CPU among many jobs running seemingly at the same time. The basic idea is simple: run one process for a little while, then run another one, and so forth. By time sharing the CPU in this manner, virtualization is achieved.
- There are a few challenges, however, in building such virtualization machinery.
    1. Performance. How can we implement virtualization without adding excessive overhead to the system?
    2. Control. How can we run processes efficiently while retaining control over the CPU? Control is particularly important to the OS, as it is in charge of resources; without control, a process could simply run forever and take over the machine, or access information that it should not be allowed to access. Obtaining high performance while maintaining control is thus one of the central challenges in building an operating system.
- Basic Technique - Limited Direct Execution:
- **Direct execution** simply means just run the program directly on the CPU. Thus, when the OS wishes to start a program running, it creates a process entry for it in a process list, allocates some memory for it, loads the program code into memory (from disk), locates its entry point, jumps to it, and starts running the user's code.
- This approach gives rise to a few problems:
    1. If we just run a program, how can the OS make sure the program doesn't do anything that we don't want it to do, while still running it efficiently?
    2. When we are running a process, how does the operating system stop it from running and switch to another process, thus implementing the time sharing we require to virtualize the CPU?
- Problem #1 - Restricted Operations:
- Direct execution has the obvious advantage of being fast; the program runs natively on the hardware CPU and thus executes as quickly as one would expect. But running on the CPU introduces a problem: what if the process wishes to perform some kind of restricted operation, such as issuing an I/O request to a disk, or gaining access to more system resources such as CPU or memory?
- A process must be able to perform I/O and some other restricted operations, but without giving the process complete control over the system.
- One approach would simply be to let any process do whatever it wants in terms of I/O and other related operations. However, doing so would prevent the construction of many kinds of systems that are desirable. For example, if we wish to build a file system that checks permissions before granting access to a file, we can't simply let any user process issue I/Os to the disk; if we did, a process could simply read or write the entire disk and thus all protections would be lost.
- Thus, the approach we take is to introduce a new processor mode, known as **user mode**. Code that runs in user mode is restricted in what it can do. For example, when running in user mode, a process can't issue I/O requests; doing so would result in the processor raising an exception; the OS would then likely kill the process.
- In contrast to user mode is **kernel mode**, which the operating system or kernel runs in. In this mode, code that runs can do what it likes, including privileged operations such as issuing I/O requests and executing all types of restricted instructions.
- To enable user processes to perform some kind of privileged operation, virtually all modern hardware provides the ability for user programs to perform a **system call**.

- System calls allow the kernel to carefully expose certain key pieces of functionality to user programs, such as accessing the file system, creating and destroying processes, communicating with other processes, and allocating more memory.
- To execute a system call, a program must execute a special **trap instruction**. This instruction simultaneously jumps into the kernel and raises the privilege level to kernel mode. Once in the kernel, the system can now perform whatever privileged operations are needed (if allowed), and thus do the required work for the calling process. When finished, the OS calls a special **return-from-trap instruction**, which returns into the calling user program while simultaneously reducing the privilege level back to user mode.
- The hardware assists the OS by providing different modes of execution. In user mode, applications do not have full access to hardware resources. In kernel mode, the OS has access to the full resources of the machine. Special instructions to trap into the kernel and return-from-trap back to user-mode programs are also provided, as well as instructions that allow the OS to tell the hardware where the **trap table** resides in memory.
- The hardware needs to be a bit careful when executing a trap, in that it must make sure to save enough of the caller's registers in order to be able to return correctly when the OS issues the return-from-trap instruction. On x86, for example, the processor will push the program counter, flags, and a few other registers onto a per-process **kernel stack**. The return-from trap will pop these values off the stack and resume execution of the usermode program.
- The kernel must carefully control what code executes upon a trap. The user/user mode can't arbitrarily jump into a memory address in the kernel. The kernel does so by setting up a trap table at boot time. When the machine boots up, it does so in kernel mode, and thus is free to configure machine hardware as needed. 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 the locations of these **trap handlers**, usually with some kind of special instruction. Once the hardware is informed, it remembers the location of these handlers until the machine is next rebooted, and thus the hardware knows what to do (i.e., what code to jump to) when system calls and other exceptional events take place.
- To specify the exact system call, a **system-call number** is usually assigned to each system call. The user code is thus responsible for placing the desired system-call number in a register or at a specified location on the stack. The OS, when handling the system call inside the **trap handler**, examines this number, ensures it is valid, and, if it is, executes the corresponding code. This level of indirection serves as a form of protection; user code cannot specify an exact address to jump to, but rather must request a particular service via number.
- <u>Problem #2 - Switching Between Processes:</u>
- The next problem with direct execution is achieving a switch between processes.
- If a program is running on the CPU, that means the OS isn't running. We need to devise a solution for the operating system to regain control of the CPU so that it can switch between processes.
- One approach that some systems have taken in the past is known as the **cooperative approach**. In this style, the OS trusts the processes of the system to behave reasonably. Processes that run for too long are assumed to periodically give up the CPU so that the OS can decide to run some other task.
- Most processes transfer control of the CPU to the OS quite frequently by making system calls. Systems like this often include an explicit yield system call, which does nothing except to transfer control to the OS so it can run other processes.

- Applications also transfer control to the OS when they do something illegal. For example, if an application divides by zero, or tries to access memory that it shouldn't be able to access, it will generate a trap to the OS. The OS will then have control of the CPU again and likely terminate the offending process.
- While the cooperative approach sounds good in theory, in practice, it is terrible. Malicious software/programs can take control of the CPU forever.
- A **non-cooperative approach** would be to use a timer. A **timer device** can be programmed to raise an interrupt every so many milliseconds. When the interrupt is raised, the currently running process is halted, and a pre-configured interrupt handler in the OS runs. At this point, the OS has regained control of the CPU.
- The OS must inform the hardware of which code to run when the timer interrupt occurs, which it does, at boot time. Also during the boot sequence, the OS must start the timer.
- Saving and Restoring Context:
- Now that the OS has regained control, whether cooperatively via a system call, or more forcefully via a timer interrupt, a decision has to be made: whether to continue running the currently-running process, or switch to a different one. This decision is made by a part of the operating system known as the **scheduler**.
- If the decision is made to switch, the OS then executes a **context switch**. A context switch is conceptually simple: all the OS has to do is save a few register values for the currently-executing process and restore a few for the soon-to-be-executing process. By doing so, the OS thus ensures that when the return-from-trap instruction is finally executed, instead of returning to the process that was running, the system resumes execution of another process.
- To save the context of the currently-running process, the OS will execute some low-level assembly code to save the general purpose registers, PC, and the kernel stack pointer of the currently-running process, and then restore said registers, PC, and switch to the kernel stack for the soon-to-be-executing process. By switching stacks, the kernel enters the call to the switch code in the context of one process (the one that was interrupted) and returns in the context of another (the soon-to-be-executing one). When the OS then finally executes a return-from-trap instruction, the soon-to-be-executing process becomes the currently-running process. And thus the context switch is complete.
- **Scheduling - Introduction:**
- Workload Assumptions:
- We will make the following assumptions about the processes, sometimes called jobs, that are running in the system:
    1. Each job runs for the same amount of time.
    2. All jobs arrive at the same time.
    3. Once started, each job runs to completion.
    4. All jobs only use the CPU (I.e. They perform no I/O).
    5. The run-time of each job is known.
- Scheduling Metrics:
- We also need a **scheduling metric**. For now, our scheduling metric will be the turnaround time.
- The **turnaround time** of a job is defined as the time at which the job completes minus the time at which the job arrived in the system.
  T_turnaround = T_completion − T_arrival
- Because we have assumed, for now, that all jobs arrive at the same time, T_arrival = 0 and hence T_turnaround = T_completion.
- You should note that turnaround time is a performance metric.

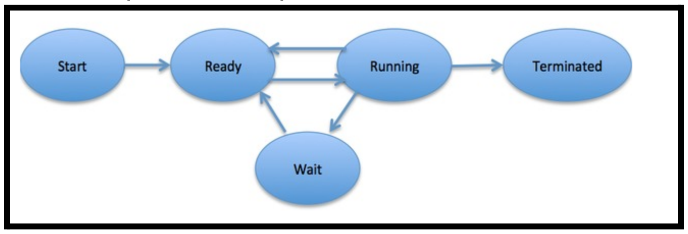
- First In, First Out (FIFO):
- The most basic algorithm we can implement is known as **First In, First Out (FIFO) scheduling** or sometimes **First Come, First Served (FCFS)**.
- FIFO has a number of positive properties: it is clearly simple and thus easy to implement.
- However, the main problem with this approach is the **convoy effect**. This occurs when a number of relatively-short potential consumers of a resource get queued behind a heavyweight resource consumer.
- Shortest Job First (SJF):
- In this approach, the shortest job is run first, then the next shortest, and so on.
- However, SJF also has a problem with the convoy effect.
- E.g. Suppose we have 3 processes, A, B and C. Suppose A arrives at t = 0 and runs for 100 seconds, B arrives at t = 10 and runs for 10 seconds and C arrives at t = 20 and runs for 10 seconds. Both B and C are blocked from running, even though they have a shorter run time than A, because they arrived later.
- Shortest Time-to-Completion First (STCF):
- To address this concern, we need to relax assumption 3 (that jobs must run to completion).
- In this approach, if a new thread arrives with a CPU burst length less than the remaining time of the current executing thread, then preempt current thread.
- A New Metric - Response Time:
- We define **response time** as the time from when the job arrives in a system to the first time it is scheduled.
- T_response = T_firstrun − T_arrival
- None of the 3 above methods (FIFO, SJF and STCF) are particularly good for response time.
- Round Robin:
- To solve this problem, we will introduce a new scheduling algorithm, classically referred to as **Round-Robin (RR)**.
- The basic idea is simple: instead of running jobs to completion, RR runs a job for a **time slice** (sometimes called a **scheduling quantum**) and then switches to the next job in the run queue. It repeatedly does so until the jobs are finished.
- Note that the length of a time slice must be a multiple of the timer-interrupt period.
- The length of the time slice is critical for RR. The shorter it is, the better the performance of RR under the response-time metric. However, making the time slice too short is problematic: suddenly the cost of context switching will dominate overall performance. Thus, deciding on the length of the time slice presents a trade-off to a system designer, making it long enough to amortize the cost of switching without making it so long that the system is no longer responsive.
- Note that the cost of context switching does not arise solely from the OS actions of saving and restoring a few registers. When programs run, they build up a great deal of state in CPU caches, TLBs, branch predictors, and other on-chip hardware. Switching to another job causes this state to be flushed and a new state relevant to the currently-running job to be brought in, which may exact a noticeable performance cost.
- RR is good with response time but bad with turnaround time.

- **Scheduling - The Multi-Level Feedback Queue (MLFQ):**
- The fundamental problem MLFQ tries to address is two-fold. First, it would like to optimize turnaround time. Second, MLFQ would like to make a system feel responsive to interactive users and thus minimize response time
- <u>MLFQ - Basic Rules:</u>
- **Rule 1:** If Priority(A) > Priority(B), A runs & B doesn't.
- **Rule 2:** If Priority(A) = Priority(B), A & B run in round-robin fashion using the time slice (quantum length) of the given queue.
- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.
- In our treatment, the MLFQ has a number of distinct queues, each assigned a different priority level. At any given time, a job that is ready to run is on a single queue. MLFQ uses priorities to decide which job should run at a given time: a job with higher priority (I.e. A job on a higher queue) is chosen to run.
- Of course, more than one job may be on a given queue, and thus have the same priority. In this case, we will just use round-robin scheduling among those jobs.
- The key to MLFQ scheduling therefore lies in how the scheduler sets priorities. Rather than giving a fixed priority to each job, MLFQ varies the priority of a job based on its observed behavior. If, for example, a job repeatedly relinquishes the CPU while waiting for input from the keyboard, MLFQ will keep its priority high, as this is how an interactive process might behave. If, instead, a job uses the CPU intensively for long periods of time, MLFQ will reduce its priority. In this way, MLFQ will try to learn about processes as they run, and thus use the history of the job to predict its future behavior.

Processes:

- A **process** is the execution of a program that allows you to perform the appropriate actions specified in a program.
- It can be defined as an execution unit where a program runs. The OS helps you to create, schedule, and terminate the processes which are used by the CPU.
- The processes created by the main process are called **child processes**.
- A process's operations can be easily controlled with the help of **PCB (Process Control Block)**.
- A PCB is a data structure maintained by the OS for every process. The PCB is identified by a process ID (PID) and contains the following information:
    1. **Process State:** The current state of the process. (I.e. Whether it is ready, running, waiting, etc.)
    2. **Process Privileges:** This is required to allow/disallow access to system resources.
    3. **Process ID (PID):** A PID is a unique identification for each of the processes in the os.
    4. **Pointer:** A pointer to the parent process.
    5. **Program Counter (PC):** The PC is a pointer to the address of the next instruction to be executed for this process.
    6. **CPU registers:**
    7. **CPU Scheduling Information:** Used to process priority and other scheduling information which are required to schedule the processes.
    8. **Accounting Information:** This includes the amount of CPU used for process execution, time limits, execution ID etc.
    9. **IO Status Information:** A list of I/O devices allocated to the process.
- A process will be in 1 of the following 5 stages at a given time:
    1. Start
    2. Ready
    3. Running
    4. Blocked/Waiting
    5. Terminated/Exited/Finished

  Here is a picture of a process's flow:



- The OS has 3 queues to keep track of processes:
    1. **Job Queue:** This queue keeps all the processes in the system.
    2. **Ready Queue:** This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
    3. **Waiting Queue:** The processes which are blocked due to unavailability of an I/O device are in this queue.
- A process can have multiple threads.

Threads:

- A **thread** is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, and thread ID.
- A thread is also called a lightweight process.
- Traditional/heavyweight processes have a single thread of control, but multi-threaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files.
- Each thread belongs to exactly one process and no thread can exist outside a process.

Context Switching:

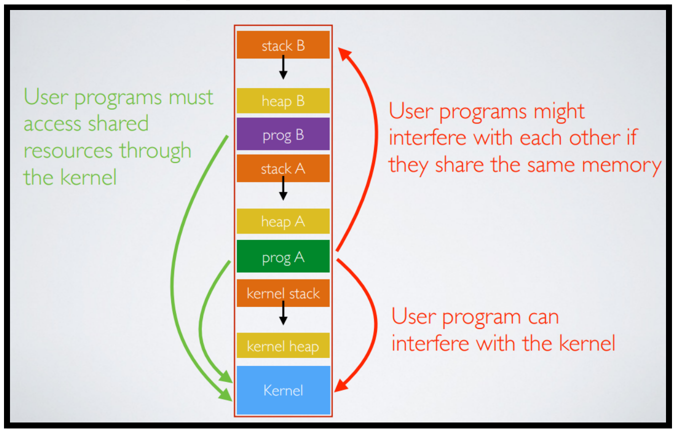
- A **context switch** is the process of storing the state of a process or thread, so that it can be restored and resume execution at a later point. This allows multiple processes to share a single CPU, and is an essential feature of a multitasking operating system.

Processes vs Threads:

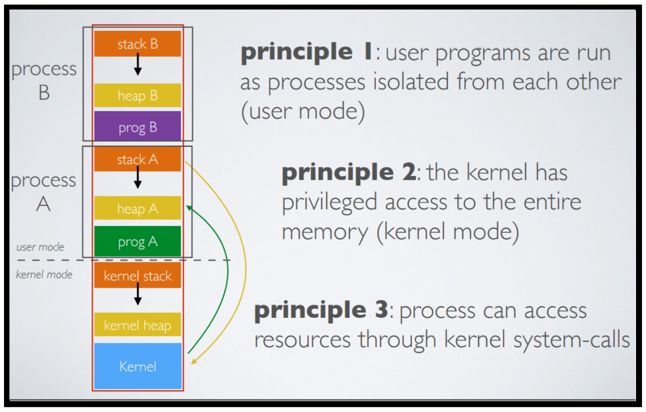
Parameter	**Processes**	**Threads**
Definition	A program in execution	A segment of a process
Termination time	Take more time to terminate	Take less time to terminate
Creation time	Take more time for creation	Take less time for creation
Communication	Communication between processes needs more time	Communication between threads requires less time
Context switching time	Take more time	Take less time
Resource	Consume more resources	Consume fewer resources
Sharing	Do not share data	Share data with each other

Lecture Notes:

- **The need for protection:**


stack B
heap B
prog B
stack A
heap A
prog A
kernel stack
kernel heap
Kernel
User programs must access shared resources through the kernel
User programs might interfere with each other if they share the same memory
User program can interfere with the kernel

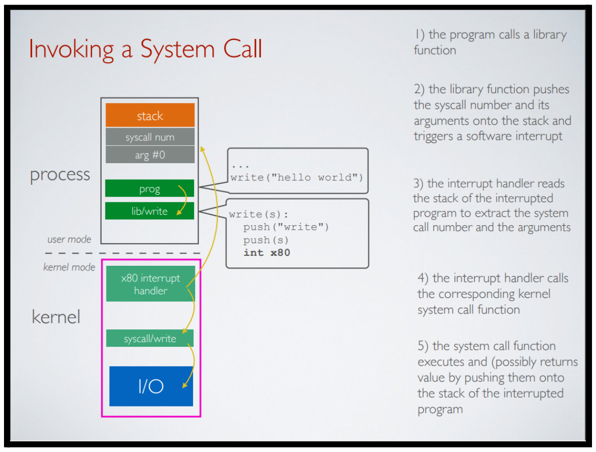
- **User mode:** User programs are run as processes isolated from each other.
- **Kernel Mode:** The kernel has privileged access to the entire memory.
- Processes can access resources through kernel system-calls.
- I.e.


process B
process A
user mode
kernel mode
stack B
heap B
prog B
stack A
heap A
prog A
kernel stack
kernel heap
Kernel
principle 1: user programs are run as processes isolated from each other (user mode)
principle 2: the kernel has privileged access to the entire memory (kernel mode)
principle 3: process can access resources through kernel system-calls

- We can use virtual memory to isolate processes and kernel memory spaces. In a nutshell, user programs do not directly access the memory but the virtual memory that is somehow mapped onto the real memory and the kernel manages the virtual memory for all processes.
- **System Call:**
- **System calls** provide user programs with an API to use the services of the operating system. Think of it as an API. System calls look like some sort of "kernel API".
- There are 6 categories of system calls:
  1. Process control
  2. File management
  3. Device management
  4. Information/maintenance (system configuration)
  5. Communication (IPC)
  6. Protection

- E.g. Suppose you are writing a Bash-like program that reads keyboard inputs from the user.

  How do you know which keyboard to listen to? There's the default keyboard, there's wireless keyboard/bluetooth keyboards, and others.
  User programs do not operate I/O devices directly. The OS abstracts those functionalities and manages access through system calls.
- To invoke a system call, you have to use **software interrupts**/**syscall trap**. This is because user programs don't have access to the kernel's memory.
- E.g.



Invoking a System Call

```
...
write("hello world")
```

```
write(s):
  push("write")
  push(s)
  int x80
```

1) the program calls a library function

2) the library function pushes the syscall number and its arguments onto the stack and triggers a software interrupt

3) the interrupt handler reads the stack of the interrupted program to extract the system call number and the arguments

4) the interrupt handler calls the corresponding kernel system call function

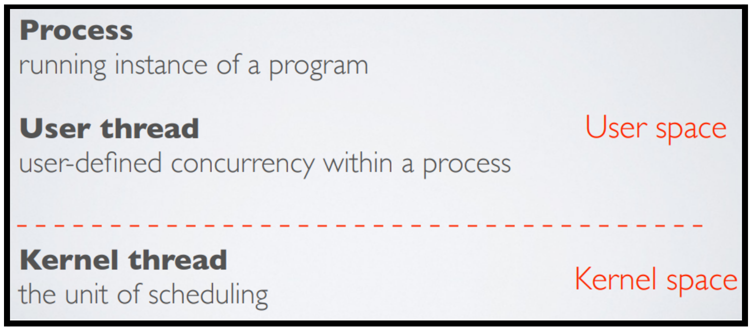
5) the system call function executes and (possibly returns value by pushing them onto the stack of the interrupted program

- **Process:**
- The **PCB (Process Control Block)** is a data structure to records process information:
  - Pid (process id) and ppid (parent process)
  - User (Optional)
  - Address space
  - Open files
  - Others
- A process is created by another process.
- The kernel creates the root process as part of the booting.
  E.g shell program for a simple OS
  E.g Window Manager for a GUI OS
- The root process has a pid of 0.
- **Process Creation on Unix:**
- The system call fork() creates a new process:
  1. Creates and initializes a new PCB.
  2. Creates a new address space.
  3. Initializes the address space with a copy of the entire contents of the address space of the parent, with one exception - the PCB.
  4. Initializes the kernel resources to point to the resources used by the parent process such as open files.
  5. Creates a kernel thread associated with this process and places that thread onto the ready queue.
- fork() is very useful when the child is cooperating with the parent and relies upon the parent's data to accomplish its task.
- Another very important system call is exec().
- **Note:** exec() does not create a new process, but rather, it runs the specified process.

- int exec(char *prog, char *argv[])
    1. Stops the current process
    2. Loads the program “prog” into the process’ address space
    3. Initializes hardware context and args for the new program
    4. Places the PCB onto the ready queue
- You use fork() and exec() together. fork() creates a new process while exec() runs the new process. Most calls to fork are followed by exec. Using fork() and exec() in combination is called **spawning**. This is actually what happens when you run Unix commands. After you type a command and hit enter, a new process will be created via fork() and then exec will run the command you typed.
- In Unix, the wait() system call makes a parent process wait for at least 1 child process to terminate.
- In Unix, the exit() system terminates a process.
- **Process Creation on Window:**
- Windows doesn’t use fork() and exec() due to security concerns. It has its own function, BOOL CreateProcess(char *prog, char *args).
- CreateProcess:
    1. Creates and initializes a new PCB
    2. Creates and initializes a new address space
    3. Loads the program specified by “prog” into the address space
    4. Copies “args” into memory allocated in address space
    5. Initializes the saved hardware context to start execution at main (or wherever specified in the file)
    6. Places the PCB on the ready queue
- WaitForSingleObject() is the Windows equivalent of wait().
- ExitProcess(int status) is the Windows equivalent of exit().
- **User thread:**
- Threads are important because:
    1. Creating a process is costly (space and time)
    2. Context switching is costly (time)
    3. Inter-process communication is costly (time)

  If you want something more lightweight, you can use **user threads**.
- Modern OSes separate the concepts of processes and threads.
- The thread defines a sequential execution stream within a process.
- The process defines the address space and general process attributes.
- A thread is bound to a single process but a process can have multiple threads.
- Benefits of threads:
    1. Responsiveness:
       An application can continue running while it waits for some events in the background.
    2. Resource sharing:
       Threads can collaborate by reading and writing the same data in memory instead of asking the OS to pass data around.
    3. Economy of time and space:
       No need to create a new PCB and switch the entire context (only the registers and the stack).
    4. Scalability in multi-processor architecture:
       The same application can run on multiple cores.

- **Multithreading Model:**
- Process vs Threads:

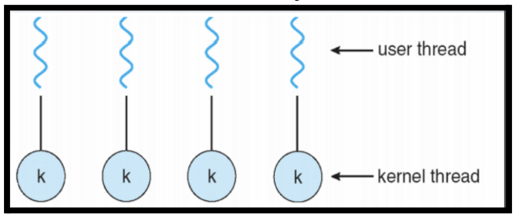


Process
running instance of a program

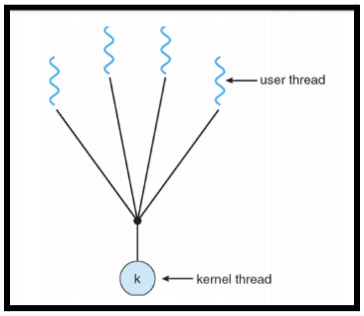
User thread User space
user-defined concurrency within a process

Kernel thread Kernel space
the unit of scheduling

- POSIX Thread APIs:
    1. Create a new thread
        - Run this function: **tid thread_create (void (*fn) (void *), void *);**
        - When you run the function, it will:
            - Allocate Thread Control Block (TCB)
            - Allocate stack
            - Put func, args on stack
            - Put thread on ready list
    2. Destroy current thread
        - Run this function: **void thread_exit ();**
    3. Wait for thread thread to exit
        - Run this function: **void thread_join (tid thread);**
- There are 3 types of multithreading models:
    1. **One-to-one model**
        - Kernel-level threads/native threads
        - Each kernel thread only has 1 user thread.



        - The kernel manages and schedules threads. Furthermore, all thread operations are managed by the kernel.
        - This way is good for scheduling but bad for speed.
    2. **Many-to-one model**
        - User-level threads/green threads
        - Each process has only 1 kernel thread, and that kernel thread can have multiple user threads.



    - Thread management and scheduling is delegated to a library, meaning that the kernel is not involved.
    - Because the kernel isn't involved, this method is very lightweight and fast but all threads can be blocked if one thread is waiting for an event and cannot be scheduled on multiple cores.

3. **Many-to-many model**
    - Hybrid threads/n:m threading
    - This model maps some N number of user threads onto some M number of kernel threads. This is a compromise between the one-to-one model and many-to-one model.
    - In general, this model is more complex to implement than the other two, because changes to both kernel and user-space code are required.
    - Here, the threading library is responsible for scheduling user threads on the available schedulable kernel threads. This makes context switching of threads very fast, as it avoids system calls.

Textbook Notes:

- **Process API:**
- The fork() System Call:
- The fork() system call is used to create a new process.
- If successful, fork() will return the pid (process id) of the child process to the parent process and 0 to the newly created child process.
- If unsuccessful, fork() will return a negative number, usually -1.
- When the child process is created, there are now two active processes in the system that we care about: the parent and the child. Assuming we are running on a system with a single CPU (for simplicity), then either the child or the parent might run at that point. Hence, with fork(), you might get a race condition.
- The CPU scheduler determines which process runs at a given moment in time. Because the scheduler is complex, we cannot usually make strong assumptions about what it will choose to do, and hence which process will run first. This nondeterminism, as it turns out, leads to some interesting problems, particularly in multi-threaded programs.
- The wait() System Call:
- A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After the child process terminates, the parent process **continues** its execution.
- The exec() System Call:
- A final and important piece of the process creation API is the exec() system call.
- This system call is useful when you want to run a program that is different from the calling program.
- The exec family of functions replaces the current running process with a new process.
- Motivating The API:
- The separation of fork() and exec() is essential in building a UNIX shell, because it lets the shell run code after the call to fork() but before the call to exec(); this code can alter the environment of the about-to-be-run program, and thus enables a variety of interesting features to be readily built.
- The shell is just a user program. It shows you a prompt and then waits for you to type something into it. You then type a command (i.e., the name of an executable program, plus any arguments) into it; in most cases, the shell then figures out where in the file system the executable resides, calls fork() to create a new child process to run the command, calls some variant of exec() to run the command, and then waits for the command to complete by calling wait(). When the child completes, the shell returns from wait() and prints out a prompt again, ready for your next command.

- <u>Process Control And Users:</u>
- Beyond fork(), exec(), and wait(), there are a lot of other interfaces for interacting with processes in UNIX systems.
- For example, the kill() system call is used to send signals to a process, including directives to pause, die, and other useful imperatives.
- The man pages contain a lot of helpful details and information.
- **Thread API:**
- <u>Thread Creation:</u>
- The first thing you have to be able to do to write a multi-threaded program is to create new threads, and thus some kind of thread creation interface must exist.
- Here's how to do it in POSIX:

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

  There are four arguments: thread, attr, start routine, and arg.
  The first, thread, is a pointer to a structure of type pthread t; we'll use this structure to interact with this thread, and thus we need to pass it to pthread_create() in order to initialize it.
  The second argument, attr, is used to specify any attributes this thread might have.
  The third argument, start_routine, is a function pointer. The function start_routine takes in a void pointer.
  Finally, the fourth argument, arg, is the argument to be passed to the function where the thread begins execution. You might ask: why do we need these void pointers? Well, the answer is quite simple: having a void pointer as an argument to the function start_routine allows us to pass in any type of argument; having it as a return value allows the thread to return any type of result.
- <u>Thread Completion:</u>
- For thread completion, you must call the function pthread_join().
  int pthread_join(pthread_t thread, void **value_ptr);
- This routine takes two arguments.
- The first is of type pthread_t, and is used to specify which thread to wait for. This variable is initialized by the thread creation routine (when you pass a pointer to it as an argument to pthread_create()). If you keep it around, you can use it to wait for that thread to terminate.
- The second argument is a pointer to the return value you expect to get back. Because the routine can return anything, it is defined to return a pointer to void; because the pthread_join() routine changes the value of the passed in argument, you need to pass in a pointer to that value, not just the value itself.

- Locks:
- Beyond thread creation and join, probably the next most useful set of functions provided by the POSIX threads library are those for providing mutual exclusion to a critical section via locks. The most basic pair of routines to use for this purpose is provided by the following:

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

- When you have a region of code that is a critical section, and thus needs to be protected to ensure correct operation, locks are quite useful.
- Here's an example:

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

The intent of the code is as follows: if no other thread holds the lock when pthread_mutex_lock() is called, the thread will acquire the lock and enter the critical section. If another thread does indeed hold the lock, the thread trying to grab the lock will not return from the call until it has acquired the lock (implying that the thread holding the lock has released it via the unlock call). Of course, many threads may be stuck waiting inside the lock acquisition function at a given time; only the thread with the lock acquired, however, should call unlock.

Unfortunately, this code is broken, in two important ways.
The first problem is a lack of proper initialization. All locks must be properly initialized in order to guarantee that they have the correct values to begin with and thus work as desired when lock and unlock are called. With POSIX threads, there are two ways to initialize locks. One way to do this is to use PTHREAD_MUTEX_INITIALIZER, as follows:
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
Doing so sets the lock to the default values and thus makes the lock usable.
The second problem with the code above is that it fails to check error codes when calling lock and unlock. Just like virtually any library routine you call in a UNIX system, these routines can also fail! If your code doesn't properly check error codes, the failure will happen silently, which in this case could allow multiple threads into a critical section.
- Condition Variables:
- The other major component of any threads library, and certainly the case with POSIX threads, is the presence of a condition variable.
- Condition variables are useful when some kind of signaling must take place between threads, if one thread is waiting for another to do something before it can continue.
- Two primary routines are used by programs wishing to interact in this way:

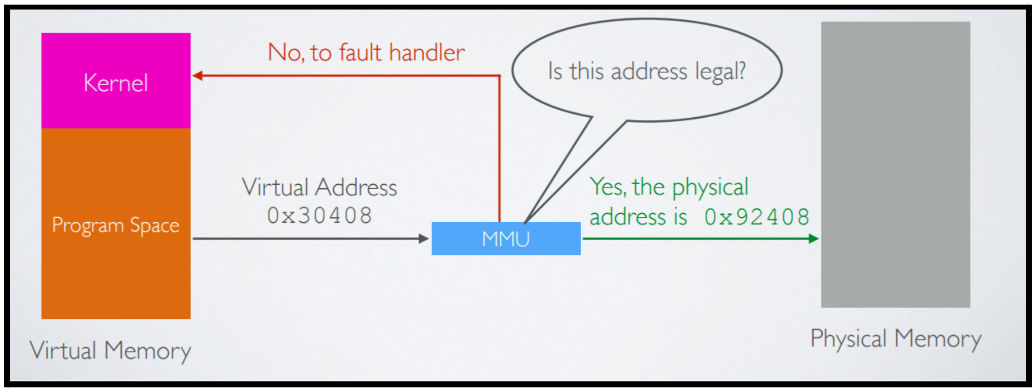
```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

- To use a condition variable, one has to have a lock that is associated with this condition. When calling either of the above routines, this lock should be held.
- The first routine, pthread_cond_wait(), puts the calling thread to sleep, and thus waits for some other thread to signal it, usually when something in the program has changed that the now-sleeping thread might care about.

Lecture Notes:

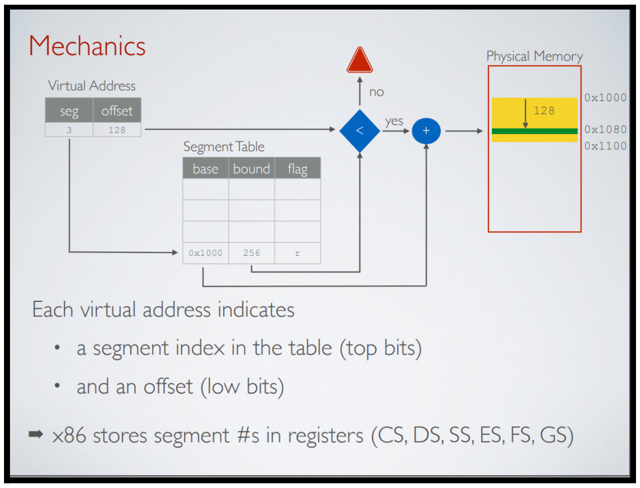
- **Virtual Memory:**
- While it is fine to place multiple execution contexts (stack and heap) at random locations in memory, having programs placed at random locations is problematic. Since function addresses and others are hard-encoded in the binary, the program cannot be placed at random locations in memory.
- A **compiler** takes source code files and translates (binds) symbolic addresses to logical, relocatable addresses within the compilation unit (object file).
- A **linker** takes a collection of object files and translates addresses to logical, absolute addresses within the executable (resolves references to symbols defined in other files/ modules).
- A naive approach is called **load time linking**. Here, we will do the linking and determine where the process will reside in memory and adjust all references within the program when the process is executed, not at compile time.
- However, this has a number of issues, such as:
    - How do we relocate the program in memory during execution? This is quite common because we're using functions and pointers.
    - What happens if there is no contiguous free region that fits the program?
    - How do we avoid programs interfering with each other?
- Issues with sharing physical memory include:
    1. **Transparency:**
    - A process shouldn't require particular physical memory bits.
    - A process often requires large amounts of contiguous memory (for stack, large data structures, etc).
    2. **Resource exhaustion:**
    - Programmers typically assume that the machine has enough memory but in reality, the sum of the sizes of all processes is often greater than physical memory.
    3. **Protection:**
    - How do we prevent A from observing B's memory?
    - How do we prevent process A from corrupting B's memory?
- We can use virtual memory to deal with these issues.
- Virtual memory goals:
    - Provide a convenient abstraction for programming by giving each program its own virtual address space.
    - Allow programs to see more memory than what exists.
    - Allocate scarce memory resources among competing processes to maximize performance with minimal overhead.
    - Enforce protection by preventing one process from messing with another's memory.
- Terminology:
    - Programs load/store to **virtual addresses**. Even the kernel.
    - Actual memory uses **physical addresses**.
    - Virtual memory hardware is called **MMU (Memory Management Unit)**. It is usually part of the CPU and is configured through privileged instructions. It translates from virtual to physical addresses and gives a per-process view of memory called the **address space**.
- The application does not see physical memory addresses. The MMU relocates each load/store at runtime. This means we can relocate processes while running either in memory or to disk.

I.e.



- Techniques for implementing virtual memory:
    1. Basic address translation
    2. Segmentation (the old way)
    3. Paging (the new way)
- **Basic Address Translation:**
- We have two special privileged registers: **base** and **bound**.
- On each load/store/jump:
    - First, set the physical address to be equal to virtual address + base
    - Then, check 0 ≤ virtual address < bound, else trap to kernel
- The OS can change these registers to move the process in memory.
- The OS must reload base for these registers on context switches. This is because each process has a different base value.
- Advantages:
    - Cheap in terms of hardware. There are only two registers.
    - Cheap in terms of cycles. Add and compare can be done in parallel.
- Disadvantages:
    - Growing a process is expensive.
    - No way to share code or data. One solution to this issue is **segmentation** (I.e. Have separate code, stack and data segments).
- **Segmentation:**
- The idea behind segmentation is that each process has a collection of multiple base/bound registers.
- This means that the address space is built from many segments (a.k.a segmentation table) and thus, it can share/protect memory at segment granularity.

- The table works like this:

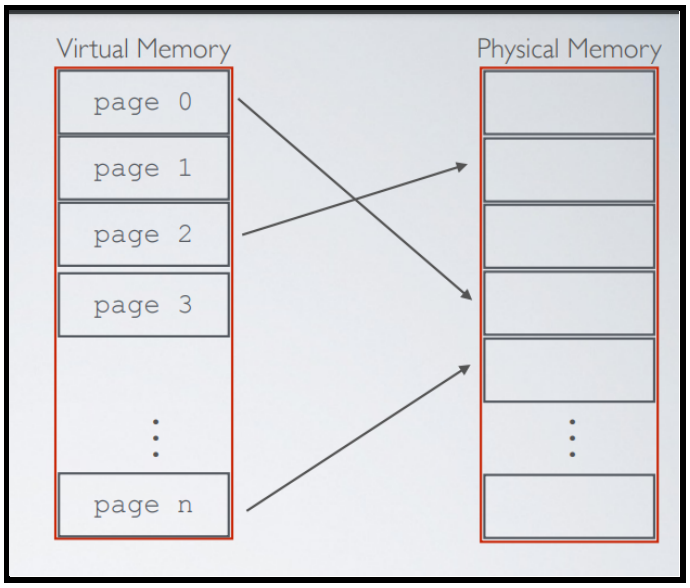


Each virtual address indicates

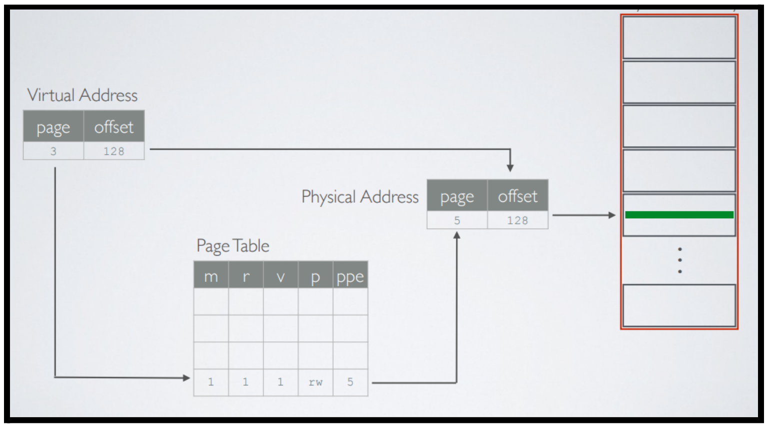
- a segment index in the table (top bits)
- and an offset (low bits)

➡ x86 stores segment #s in registers (CS, DS, SS, ES, FS, GS)

- Advantages:
    - Multiple segments per process (sparse memory).
    - Can easily share memory.
    - Do not need the entire process in memory (swap).
- Disadvantages:
    - Requires translation, which could limit performance.
    - Makes external **fragmentation** a real problem. Basically, you have a bunch of small chunks of free memory that can't be used.
- Fragmentation is the inability to use free memory.
- **External fragmentation** occurs because of variable sized pieces (Many small holes).
- **Internal fragmentation** occurs because of fixed size pieces (I.e. There are no external holes but an internal waste of space)
- **Paging:**
- The idea of paging is to divide the memory up into fixed-size pages, usually 4kb, to eliminate external fragmentation. Furthermore, each process has a collection of maps from virtual pages to physical pages. This can share/protect memory at page granularity. E.g.



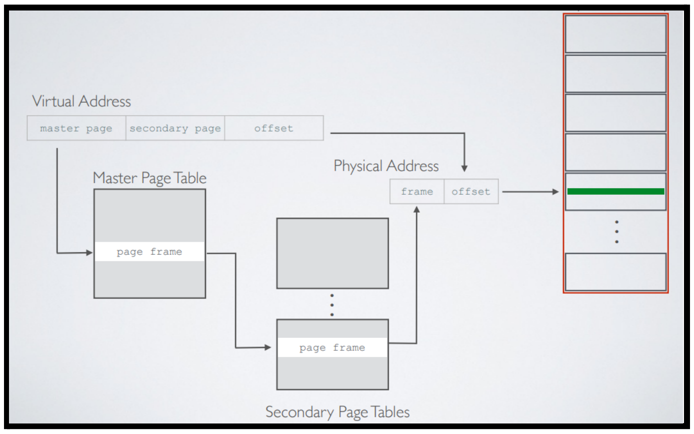
- Paging eliminates external fragmentation and simplifies allocation, free, and backing storage (swap). However, paging does cause internal fragmentation. The average internal fragmentation is .5 pages per "segment".
- Pages are fixed size (e.g. 4K) so a virtual address has two parts:
    1. **Virtual page number**, which is the most significant bits
    2. **Page offset**, which is the least significant 12 bits ($\log_2 4k$)
- The page table is a collection of **page table entry (PTE)** that maps a **virtual page number (VPN)**, which is the index in the page table, to physical page numbers (PPN), which is also called frame number, and includes bits for protection, validity, etc.
- The **Modify bit** says whether or not the page has been written (set when the write to a page occurs).
- The **Reference bit** says whether the page has been accessed (set when a read or write to a page occurs).
- The **Valid bit** says whether or not the PTE can be used (checked each time the virtual address is used).
- The **Protection bits** say what operations (read, write, execute) are allowed on page.
- The **Physical page number (PPN)** determines the physical page.
- Page Lookup Example:



- Advantages:
    - Easy to allocate memory:
        - Memory comes from a free list of fixed size chunks.
        - Allocating a page is just removing it from the list.
        - External fragmentation is not a problem.
    - Easy to swap out chunks of a program:
        - All chunks are the same size.
        - Use a valid bit to detect references to swapped pages.
        - Pages are a convenient multiple of the disk block size.
- Disadvantages:
    - Can still have internal fragmentation.
    - Requires 2 or more references, which could limit performance.
      The solution is to use a hardware cache of lookups.
    - The amount of memory to store the page table is significant.
      Need one PTE per page, with 32 bit address space w/ 4KB pages = 2^20 PTEs.
      4 bytes/PTE = 4MB/page table
      25 processes = 100MB just for page tables
      The solution is to page the page tables.

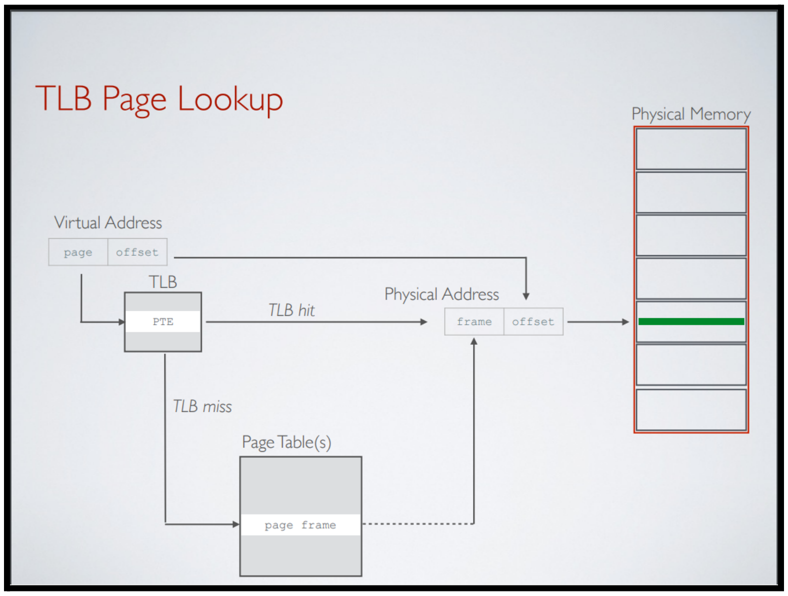
- **Improving Paging:**
    - Smaller page tables
    - Faster address translation
    - Larger virtual than physical memory (swapping)
    - Advanced Functionality
- **Smaller Page Tables:**
- Each process has a page table defining its address space.
- Considering 32-bit address space with 4K the size of the pages table is $2^{32} / 2^{12} \times 4B = 4MB$ / process this is a big overhead.
- The problem is that the page table is not part of the data structure and needs a lot of allocated memory. We need to make it sparse. The solution is that we only need to map the portion of the address space actually being used. Hence, we can use another level of indirection: **two-level page tables**.
- Virtual addresses in 2-level page tables have three parts:
    1. **Master page number**, which is the index in the master page table that maps to a secondary page table.
    2. **Secondary page number**, which is the index in the secondary page table that maps to the physical memory.
    3. **Offset** that indicates where the physical page address is located.

  I.e.



- **Faster Address Translation:**
- Translations take a lot of time, and with two-level page tables, we now have to do the translations twice.
- One-page table : one table lookup + one fetch
- Two-page table (32 bits) : 2 table lookups + one fetch
- 4-page table (64 bits) : 4 table lookup + one fetch
- A solution is to use a **Translation Lookaside Buffer (TLB)** which caches translations in hardware to reduce lookup cost.
- TLBs are special hardwares used to translate virtual page numbers into PTEs (not physical address) in a single machine cycle.
- It is typically 4-way to fully associative cache and all entries are looked up in parallel.
- It caches 32-128 PTE values (128-512K memory).
- TLBs exploit locality. Processes only use a handful of pages at a time so the TLB hit rate is very important for performances.
- 99% of the time, you hit the TLB and not the page table.

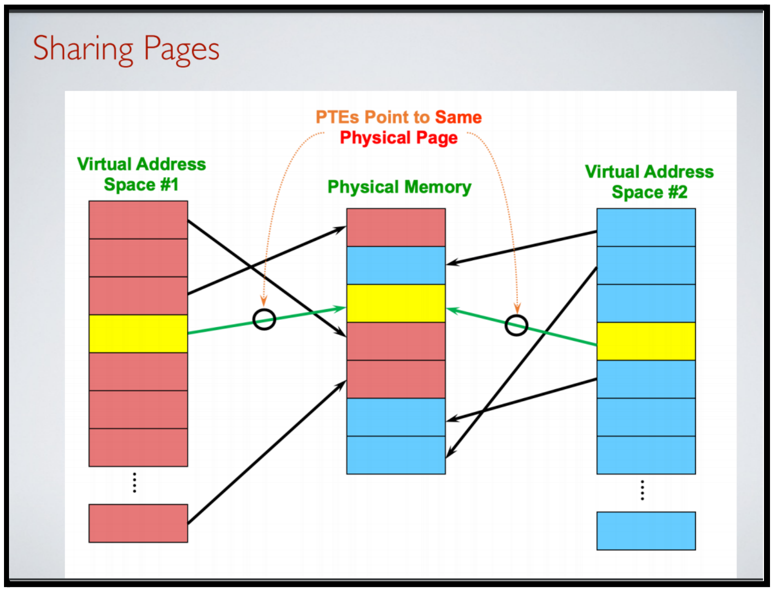
- TLB Page Lookup Example:



- Page lookup Steps:
  A process is executing on the CPU, and it issues a read to an address. The read goes to the TLB in the MMU.
  1. TLB does a lookup using the page number of the address.
  2. Common case is that the page number matches, returning a page table entry (PTE) for the mapping for this address.
  3. TLB validates that the PTE protection allows reads (in this example).
  4. PTE specifies which physical frame holds the page.
  5. MMU combines the physical frame and offset into a physical address.
  6. MMU then reads from that physical address, and returns value to the CPU.

  This is all done by the hardware.
- TLB misses can occur in 2 ways:
  1. TLB does not have a PTE mapping this virtual address.
  2. PTE in TLB, but memory access violates PTE protection bits.
- **Swapping:**
- The OS can use a disk to simulate a larger virtual memory than the physical memory meaning that pages can be moved between memory and disk (paging in/out).
- Paging process over time:
  - Initially, pages are allocated from memory.
  - When memory fills up, allocating a page requires some other page to be evicted.
  - Evicted pages go to disk, more precisely to the swap file/backing store.
  - Done by the OS, and transparent to the application.
- **Demand paging** is a method of virtual memory management. In a system that uses demand paging, the operating system copies a disk page into physical memory only if an attempt is made to access it and that page is not already in memory. It follows that a process begins execution with none of its pages in physical memory, and many page faults will occur until most of a process's working set of pages are located in physical memory. This is an example of a lazy loading technique.
- **Page Faults:**
- Read/write/execute protection bits are used to check and ensure some operations are not permitted on page. The TLB sends traps to the OS and the OS usually will send fault back up to the process.

- There are 2 possible reasons for invalid bits:
    1. Virtual page is not allocated:
        - The TLB sends traps to the OS and the OS sends a fault to the process. Then the software (OS) takes over.
    2. Virtual page is not allocated in the address space but is swapped on disk:
        - The TLB sends traps to the OS and the OS allocates a frame, reads from disk, and maps the PTE to a physical frame.
- Page fault steps:
    1. When the OS evicts a page, it sets the PTE as invalid and stores the location of the page in the swap file in the PTE.
    2. When a process accesses the page, the invalid PTE causes a trap (page fault).
    3. The trap will run the OS page fault handler.
    4. Handler uses the invalid PTE to locate the page in the swap file.
    5. Reads the page into a physical frame, updates PTE to point to it.
    6. Restarts the process.
- **Sharing:**
- Private VM spaces protect applications from each other but this makes it difficult to share data between processes. A solution is to have shared memory to allow processes to share data using direct memory references. This requires synchronization.
- E.g.



- Can we map shared memory at the same or different virtual addresses in each process' address space?
    - **Different Mapping:** Flexible but pointers inside shared memory are invalid.
    - **Same Mapping:** Less flexible but shared pointers are valid.

- **Copy on Write:**
- OSes spend a lot of time copying data.
  Here are 2 examples:
    1. System call arguments between user/kernel space.
    2. Entire address spaces to implement fork().
- Use **Copy on Write (CoW)** to defer large copies as long as possible, hoping to avoid them altogether.
    - Create shared mappings of parent pages in child virtual address space (instead of copying pages).
    - Shared pages are protected as read-only in parent and child.
      Any write operation will generate a protection fault, send a trap to the OS, copy the page, change page mapping in client page table, and restart write instruction.
- **Mapped Files:**
- **Mapped files** enable processes to do file I/O using loads and stores.
- Instead of "open, read into buffer, operate on buffer, etc" we bind a file to a virtual memory region.
    - PTEs map virtual addresses to physical frames holding file data.
    - Virtual address base + N refers to offset N in file.
- Initially, all pages mapped to file are invalid, similar to a swapped page.
    - The OS reads a page from a file when invalid page is accessed.
    - The OS writes a page to file when evicted, or region unmapped.
    - If the page is not dirty (has not been written to), no write is needed (another use of the dirty bit in PTE).
- Advantages:
    - Uniform access for files and memory (just use pointers).
- Disadvantages:
    - Process has less control over data movement as the OS handles faults transparently.
    - Does not generalize to streamed I/O (pipes, sockets, etc).
- **x86 architecture supports both paging and segmentation:**
- x86 architecture supports both paging and segmentation.

Textbook Notes:

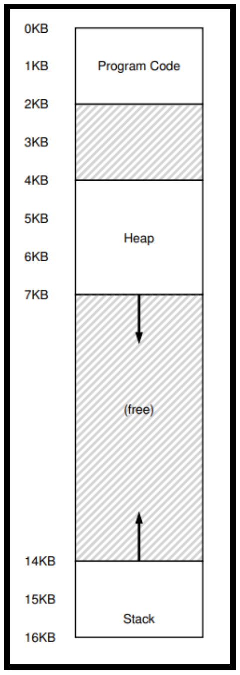
- **Mechanism - Address Translation:**
- Introduction:
- In developing the virtualization of the CPU, we focused on a general mechanism known as **limited direct execution (LDE)**.
- The idea behind LDE is simple. For the most part, let the program run directly on the hardware; however, at certain key points in time (such as when a process issues a system call, or a timer interrupt occurs), arrange so that the OS gets involved and makes sure the "right" thing happens. Thus, the OS, with a little hardware support, tries its best to get out of the way of the running program, to deliver an efficient virtualization. However, by interposing at those critical points in time, the OS ensures that it maintains control over the hardware. Efficiency and control together are two of the main goals of any modern operating system.
- **Interposition** is a generic and powerful technique that is often used to great effect in computer systems. In virtualizing memory, the hardware will interpose on each memory access, and translate each virtual address issued by the process to a physical address where the desired information is actually stored. However, the general technique of interposition is much more broadly applicable; indeed, almost any well-defined interface can be interposed upon, to add new functionality or improve some other aspect of the system. One of the usual benefits of such an approach is transparency; the interposition

often is done without changing the interface of the client, thus requiring no changes to said client.

- In virtualizing memory, we will pursue a similar strategy, attaining both efficiency and control while providing the desired virtualization. Efficiency dictates that we make use of hardware support, which at first will be quite rudimentary but will grow to be fairly complex.
- The generic technique we will use, which you can consider an addition to our general approach of limited direct execution, is something that is referred to as **hardware-based address translation/address translation**.
- With address translation, the hardware transforms each memory access such as an instruction fetch, load, or store, changing the virtual address provided by the instruction to a physical address where the desired information is actually located. Thus, on each and every memory reference, an address translation is performed by the hardware to redirect application memory references to their actual locations in memory.
- Of course, the hardware alone cannot virtualize memory, as it just provides the low-level mechanism for doing so efficiently. The OS must get involved at key points to set up the hardware so that the correct translations take place. It must thus manage memory, keeping track of which locations are free and which are in use, and judiciously intervening to maintain control over how memory is used.
- Assumptions:
- For now, we will assume these 3 things:
  1. The user's address space must be placed contiguously in physical memory.
  2. The size of the address space is not too big, specifically, that it is less than the size of physical memory.
  3. Each address space is exactly the same size.
- Dynamic (Hardware-based) Relocation:
- Also called **base and bounds**.
- For this implementation, we'll need two hardware registers within each CPU: one is called the **base register,** and the other the **bounds** (sometimes called a **limit register**).
- This base-and-bounds pair is going to allow us to place the address space anywhere we'd like in physical memory, and do so while ensuring that the process can only access its own address space.
- In this setup, each program is written and compiled as if it is loaded at address zero. However, when a program starts running, the OS decides where in physical memory it should be loaded and sets the base register to that value. Now, when any memory reference is generated by the process, it is translated by the processor in the following manner:
  physical address = virtual address + base
- Each memory reference generated by the process is a **virtual address**. The hardware in turn adds the contents of the **base register** to this address and the result is a **physical address** that can be issued to the memory system.
- With dynamic relocation, a little hardware goes a long way. Namely, a base register is used to transform virtual addresses (generated by the program) into physical addresses. A bounds or limit register ensures that such addresses are within the confines of the address space. Together they provide a simple and efficient virtualization of memory.
- Transforming a virtual address into a physical address is exactly the technique we refer to as **address translation**. This means the hardware takes a virtual address the process thinks it is referencing and transforms it into a physical address which is where the data actually resides. Because this relocation of the address happens at runtime, and because we can move address spaces even after the process has started running, the technique is often referred to as dynamic relocation.

- The bounds register is there to help with protection. Specifically, the processor will first check that the memory reference is within bounds to make sure it is legal.
- Bound registers can be defined in one of two ways:
    1. It holds the size of the address space, and thus the hardware checks the virtual address against it first before adding the base.
    2. It holds the physical address of the end of the address space, and thus the hardware first adds the base and then makes sure the address is within bounds.
- The CPU must be able to generate exceptions in situations where a user program tries to access memory illegally (with an address that is “out of bounds”). The OS handler can then figure out how to react, in this case likely terminating the process.
- The base and bounds registers are hardware structures kept on the chip (one pair per CPU). Sometimes people call the part of the processor that helps with address translation the **memory management unit (MMU)**.
- The OS must track which parts of free memory are not in use, so as to be able to allocate memory to processes. Many different data structures can of course be used for such a task. The simplest is a **free list**, which simply is a list of the ranges of the physical memory which are not currently in use.
- Operating System Issues:
- Just as the hardware provides new features to support dynamic relocation, the OS now has new issues it must handle; the combination of hardware support and OS management leads to the implementation of a simple virtual memory. Specifically, there are a few critical junctures where the OS must get involved to implement our base-and-bounds version of virtual memory.
- First, the OS must take action when a process is created, finding space for its address space in memory. Fortunately, given our assumptions that each address space is
  (a) smaller than the size of physical memory and
  (b) the same size, this is quite easy for the OS.
  it can simply view physical memory as an array of slots, and track whether each one is free or in use. When a new process is created, the OS will have to search through the free list to find room for the new address space and then mark it used.
- The OS must do some work when a process is terminated, specifically reclaiming all of its memory for use in other processes or the OS. Upon termination of a process, the OS thus puts its memory back on the free list, and cleans up any associated data structures as need be.
- The OS must also perform a few additional steps when a context switch occurs. There is only one base and bounds register pair on each CPU, after all, and their values differ for each running program, as each program is loaded at a different physical address in memory. Thus, the OS must save and restore the base-and-bounds pair when it switches between processes. Specifically, when the OS decides to stop running a process, it must save the values of the base and bounds registers to memory or sometimes in the process’s PCB. Similarly, when the OS resumes a running process or runs it the first time, it must set the values of the base and bounds on the CPU to the correct values for this process.
- We should note that when a process is stopped, it is possible for the OS to move an address space from one location in memory to another rather easily. To move a process’s address space, the OS first deschedules the process; then, the OS copies the address space from the current location to the new location; finally, the OS updates the saved base register in the process structure to point to the new location. When the process is resumed, its new base register is restored, and it begins running again, oblivious that its instructions and data are now in a completely new spot in memory.

- Lastly, the OS must provide exception handlers. The OS installs these handlers at boot time via privileged instructions.
- **Segmentation:**
- Segmentation - Generalized Base/Bounds:
- So far we have been putting the entire address space of each process in memory. With the base and bounds registers, the OS can easily relocate processes to different parts of physical memory. However, you might have noticed something interesting about these address spaces of ours: there is a big chunk of “free” space right in the middle, between the stack and the heap. Although the space between the stack and heap is not being used by the process, it is still taking up physical memory when we relocate the entire address space somewhere in physical memory; thus, the simple approach of using a base and bounds register pair to virtualize memory is wasteful. It also makes it quite hard to run a program when the entire address space doesn’t fit into memory; thus, base and bounds are not as flexible as we would like.
- I.e.



- To solve this problem, an idea was born, and it is called **segmentation**.
- The idea is simple: instead of having just one base and bounds pair in our MMU, why not have a base and bounds pair per logical segment of the address space? A segment is just a contiguous portion of the address space of a particular length, and in our canonical address space, we have three logically-different segments: code, stack, and heap. What segmentation allows the OS to do is to place each one of those segments in different parts of physical memory, and thus avoid filling physical memory with unused virtual address space.

- E.g. As you can see in the diagram, only used memory is allocated space in physical memory, and thus large address spaces with large amounts of unused address space, called **sparse address spaces**, can be accommodated.

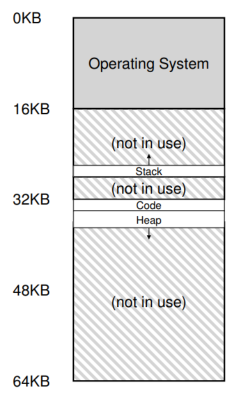


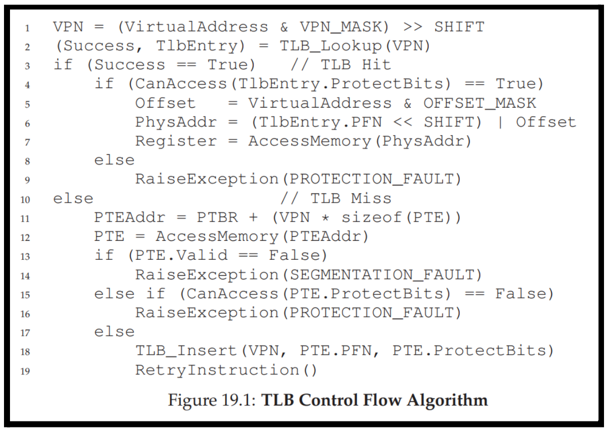
Figure 16.2: **Placing Segments In Physical Memory**

- The term **segmentation fault** arises from a memory access on a segmented machine to an illegal address.
- Which Segment Are We Referring To?:
- The hardware uses segment registers during translation, but how does it know the offset into a segment, and to which segment an address refers?
- One common approach, sometimes referred to as an **explicit approach**, is to chop up the address space into segments based on the top few bits of the virtual address.
- There are other ways for the hardware to determine which segment a particular address is in. In the **implicit approach**, the hardware determines the segment by noticing how the address was formed.
- Support for Sharing:
- As support for segmentation grew, system designers soon realized that they could realize new types of efficiencies with a little more hardware support. Specifically, to save memory, sometimes it is useful to share certain memory segments between address spaces. **Code sharing** is common and still in use in systems today.
- To support sharing, we need a little extra support from the hardware, in the form of **protection bits**.
- Basic support adds a few bits per segment, indicating whether or not a program can read or write a segment, or perhaps execute code that lies within the segment. By setting a code segment to read-only, the same code can be shared across multiple processes, without worry of harming isolation; while each process still thinks that it is accessing its own private memory, the OS is secretly sharing memory which cannot be modified by the process, and thus the illusion is preserved.
- Fine-grained vs. Coarse-grained Segmentation:
- Most of our examples thus far have focused on systems with just a few segments (code, stack, heap). We can think of this segmentation as **coarse-grained**, as it chops up the address space into relatively large, coarse chunks. However, some early systems were more flexible and allowed for address spaces to consist of a large number of smaller segments, referred to as **fine-grained segmentation**.
- Supporting many segments requires even further hardware support, with a **segment table** of some kind stored in memory. Such segment tables usually support the creation of a very large number of segments, and thus enable a system to use segments in more flexible ways.

- <u>OS Support:</u>
- You now should have a basic idea as to how segmentation works. Pieces of the address space are relocated into physical memory as the system runs, and thus a huge savings of physical memory is achieved relative to our simpler approach with just a single base/bounds pair for the entire address space. Specifically, all the unused space between the stack and the heap need not be allocated in physical memory, allowing us to fit more address spaces into physical memory and support a large and sparse virtual address space per process.
- A general problem that arises is that physical memory quickly becomes full of little holes of free space, making it difficult to allocate new segments, or to grow existing ones. We call this problem **external fragmentation**.
- One solution to this problem would be to **compact physical memory** by rearranging the existing segments. For example, the OS could stop whichever processes are running, copy their data to one contiguous region of memory, change their segment register values to point to the new physical locations, and thus have a large free extent of memory with which to work. By doing so, the OS enables the new allocation request to succeed. However, compaction is expensive, as copying segments is memory-intensive and generally uses a fair amount of processor time.
- **Paging - Introduction:**
- <u>A Simple Example And Overview:</u>
- It is sometimes said that the operating system takes one of two approaches when solving most any space-management problem. The first approach is to chop things up into variable-sized pieces, as we saw with **segmentation** in virtual memory. Unfortunately, this solution has inherent difficulties. In particular, when dividing a space into different-size chunks, the space itself can become fragmented, and thus allocation becomes more challenging over time. Thus, it may be worth considering the second approach: to chop up space into fixed-sized pieces. In virtual memory, we call this idea **paging**.
- Instead of splitting up a process's address space into some number of variable-sized logical segments (code, heap, stack), we divide it into fixed-sized units, each of which we call a **page**. Correspondingly, we view physical memory as an array of fixed-sized slots called **page frames**. Each of these frames can contain a single virtual-memory page.
- The most important improvement of paging will be flexibility. With a fully-developed paging approach, the system will be able to support the abstraction of an address space effectively, regardless of how a process uses the address space.
- Another advantage is the simplicity of free-space management that paging affords.
- To record where each virtual page of the address space is placed in physical memory, the operating system usually keeps a per-process data structure known as a **page table**. The major role of the page table is to store address translations for each of the virtual pages of the address space, thus letting us know where in physical memory each page resides.
- <u>Where Are Page Tables Stored?:</u>
- Page tables can get terribly large, much bigger than the small segment table or base/bounds pair we have discussed previously.
- Because page tables are so big, we don't keep any special on-chip hardware in the MMU to store the page table of the currently-running process. Instead, we store the page table for each process in memory.
- <u>What's Actually In The Page Table?:</u>
- The page table is just a data structure that is used to map virtual addresses (or virtual page numbers) to physical addresses (physical frame numbers).

- The simplest form is called a **linear page table**, which is just an array. The OS indexes the array by the virtual page number (VPN), and looks up the page-table entry (PTE) at that index in order to find the desired physical frame number (PFN).
- As for the contents of each PTE, we have a number of different bits in there worth understanding at some level.
- A **valid bit** is common to indicate whether the particular translation is valid; for example, when a program starts running, it will have code and heap at one end of its address space, and the stack at the other. All the unused space in-between will be marked invalid, and if the process tries to access such memory, it will generate a trap to the OS which will likely terminate the process. Thus, the valid bit is crucial for supporting a sparse address space; by simply marking all the unused pages in the address space invalid, we remove the need to allocate physical frames for those pages and thus save a great deal of memory.
- We also might have **protection bits**, indicating whether the page could be read from, written to, or executed from. Again, accessing a page in a way not allowed by these bits will generate a trap to the OS.
- A **present bit** indicates whether this page is in physical memory or on disk.
- A **dirty bit** is also common, indicating whether the page has been modified since it was brought into memory.
- A **reference bit/accessed bit** is sometimes used to track whether a page has been accessed, and is useful in determining which pages are popular and thus should be kept in memory.
- Paging - Also Too Slow:
- With page tables in memory, we already know that they might be too big. As it turns out, they can slow things down too.
- **Paging - Faster Translations (TLBs):**
- TLB Basic Algorithm:
- Using paging as the core mechanism to support virtual memory can lead to high performance overheads. By chopping the address space into small, fixed-sized units (pages), paging requires a large amount of mapping information. Because that mapping information is generally stored in physical memory, paging logically requires an extra memory lookup for each virtual address generated by the program. Going to memory for translation information before every instruction fetch or explicit load or store is prohibitively slow.
- When we want to make things fast, the OS usually needs some help. And help often comes from the OS's old friend: the hardware. To speed address translation, we are going to add what is called a **translation-lookaside buffer** or **TLB**.
- A TLB is part of the chip's memory-management unit (MMU), and is simply a hardware cache of popular virtual-to-physical address translations.
- Upon each virtual memory reference, the hardware first checks the TLB to see if the desired translation is held therein. If so, the translation is performed quickly without having to consult the page table. Because of their tremendous performance impact, TLBs in a real sense make virtual memory possible.

- Here is a basic algorithm for TLBs:



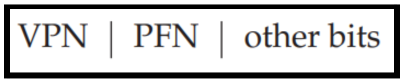
```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)    // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset   = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register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

Figure 19.1: **TLB Control Flow Algorithm**

  The algorithm the hardware follows works like this: first, extract the virtual page number (VPN) from the virtual address (Line 1), and check if the TLB holds the translation for this VPN (Line 2). If it does, we have a **TLB hit**, which means the TLB holds the translation. We can now extract the page frame number (PFN) from the relevant TLB entry, concatenate that onto the offset from the original virtual address, and form the desired physical address (PA), and access memory (Lines 5–7), assuming protection checks do not fail (Line 4). If the CPU does not find the translation in the TLB (a **TLB miss**), we have some more work to do. In this example, the hardware accesses the page table to find the translation (Lines 11–12), and, assuming that the virtual memory reference generated by the process is valid and accessible (Lines 13, 15), updates the TLB with the translation (Line 18). These sets of actions are costly, primarily because of the extra memory reference needed to access the page table (Line 12). Finally, once the TLB is updated, the hardware retries the instruction; this time, the translation is found in the TLB, and the memory reference is processed quickly.
- The TLB, like all caches, is built on the premise that in the common case, translations are found in the cache. If so, little overhead is added, as the TLB is found near the processing core and is designed to be quite fast. When a miss occurs, the high cost of paging is incurred; the page table must be accessed to find the translation, and an extra memory reference (or more, with more complex page tables) results. If this happens often, the program will likely run noticeably more slowly; memory accesses, relative to most CPU instructions, are quite costly, and TLB misses lead to more memory accesses. Thus, it is our hope to avoid TLB misses as much as we can.
- Caching is one of the most fundamental performance techniques in computer systems. The idea behind hardware caches is to take advantage of locality in instruction and data references.
- There are usually two types of locality: **temporal locality** and **spatial locality**. With temporal locality, the idea is that an instruction or data item that has been recently accessed will likely be re-accessed soon in the future. Think of loop variables or instructions in a loop; they are accessed repeatedly over time. With spatial locality, the idea is that if a program accesses memory at address x, it will likely soon access memory near x. Imagine here streaming through an array of some kind, accessing one element and then the next. Of course, these properties depend on the exact nature of the program, and thus are not hard-and-fast laws but more like rules of thumb.
- Hardware caches, whether for instructions, data, or address translations take advantage of locality by keeping copies of memory in small, fast on-chip memory. Instead of having to go to memory to satisfy a request, the processor can first check if a nearby copy exists in a cache; if it does, the processor can access it quickly and avoid spending the

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

- Who Handles The TLB Miss?:
- Two answers are possible: the hardware or the OS.
- In the olden days, the hardware had complex instruction sets, called **CISC** for **complex-instruction set computers**, and the people who built the hardware didn't much trust the OS people. Thus, the hardware would handle the TLB miss entirely. To do this, the hardware has to know exactly where the page tables are located in memory via a page table base register, as well as their exact format. On a miss, the hardware would "walk" the page table, find the correct page-table entry and extract the desired translation, update the TLB with the translation, and retry the instruction.
- More modern architectures have what is known as a software-managed TLB. On a TLB miss, the hardware simply raises an exception, which pauses the current instruction stream, raises the privilege level to kernel mode, and jumps to a trap handler. This trap handler is code within the OS that is written with the express purpose of handling TLB misses. When run, the code will lookup the translation in the page table, use special "privileged" instructions to update the TLB, and return from the trap; at this point, the hardware retries the instruction resulting in a TLB hit.
- TLB Contents - What's In There?:
- Let's look at the contents of the hardware TLB in more detail. A typical TLB might have 32, 64, or 128 entries and be what is called fully associative. Basically, this just means that any given translation can be anywhere in the TLB, and that the hardware will search the entire TLB in parallel to find the desired translation. A TLB entry might look like this:



VPN	PFN	other bits

- Note that both the VPN and PFN are present in each entry, as a translation could end up in any of these locations. The hardware searches the entries in parallel to see if there is a match.
- In other bits, there are usually valid bits, protection bits, dirty bits, etc.
- TLB Issue - Context Switches:
- With TLBs, some new issues arise when switching between processes and hence address spaces. Specifically, the TLB contains virtual-to-physical translations that are only valid for the currently running process. These translations are not meaningful for other processes. As a result, when switching from one process to another, the hardware or OS or both must be careful to ensure that the about-to-be-run process does not accidentally use translations from some previously run process.
- One approach is to simply **flush** the TLB on context switches, thus emptying it before running the next process. On a software-based system, this can be accomplished with an explicit and privileged hardware instruction; with a hardware-managed TLB, the flush could be enacted when the page-table base register is changed. In either case, the flush operation simply sets all valid bits to 0, essentially clearing the contents of the TLB. By flushing the TLB on each context switch, we now have a working solution, as a process will never accidentally encounter the wrong translations in the TLB. However, there is a cost: each time a process runs, it must incur TLB misses as it touches its data and code pages. If the OS switches between processes frequently, this cost may be high. To

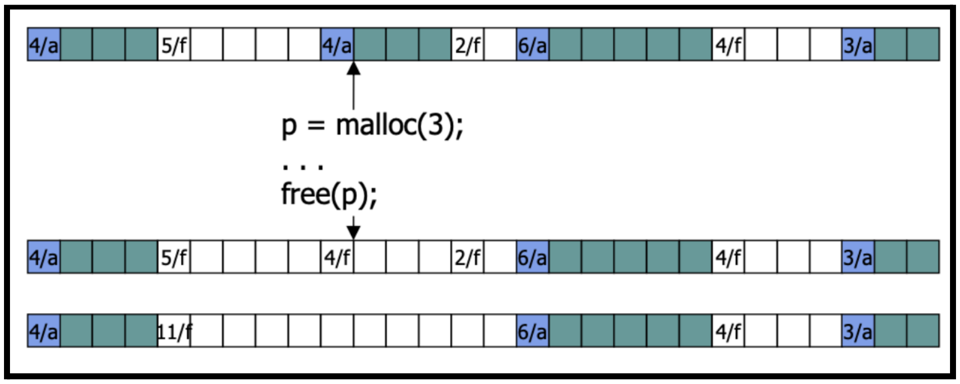
reduce this overhead, some systems add hardware support to enable sharing of the TLB across context switches. In particular, some hardware systems provide an **address space identifier (ASID)** field in the TLB. You can think of the ASID as a process identifier (PID), but usually it has fewer bits.

- Issue - Replacement Policy:
- As with any cache, and thus also with the TLB, one more issue that we must consider is **cache replacement**. Specifically, when we are installing a new entry in the TLB, we have to replace an old one, and thus the question: which one to replace?
- One common approach is to evict the **least-recently-used** or **LRU entry**. LRU tries to take advantage of locality in the memory-reference stream, assuming it is likely that an entry that has not recently been used is a good candidate for eviction.
- Another typical approach is to use a **random policy**, which evicts a TLB mapping at random. Such a policy is useful due to its simplicity and ability to avoid corner-case behaviors.
- **Paging - Smaller Tables:**
- Simple Solution - Bigger Pages:
- We now tackle the second problem that paging introduces: page tables are too big and thus consume too much memory.
- We could reduce the size of the page table in one simple way: use bigger pages.
- The major problem with this approach, however, is that big pages lead to waste within each page, a problem known as **internal fragmentation**. Applications thus end up allocating pages but only using little bits and pieces of each, and memory quickly fills up with these overly-large pages.
- Hybrid Approach - Paging and Segments:
- Another solution is to combine paging and segmentation in order to reduce the memory overhead of page tables.
- Multi-level Page Tables:
- A different approach doesn't rely on segmentation but attacks the same problem: how to get rid of all those invalid regions in the page table instead of keeping them all in memory? We call this approach a **multi-level page table**, as it turns the linear page table into something like a tree. This approach is so effective that many modern systems employ it.
- The basic idea behind a multi-level page table is simple. First, chop up the page table into page-sized units. Then, if an entire page of page-table entries (PTEs) is invalid, don't allocate that page of the page table at all. To track whether a page of the page table is valid and if valid, where it is in memory, use a new structure, called the **page directory**. The page directory thus either can be used to tell you where a page of the page table is, or that the entire page of the page table contains no valid pages.
- The page directory, in a simple two-level table, contains one entry per page of the page table. It consists of a number of page directory entries (PDE). A PDE minimally has a valid bit and a page frame number , similar to a PTE. However, as hinted at above, the meaning of this valid bit is slightly different: if the PDE is valid, it means that at least one of the pages of the page table that the entry points to is valid. If the PDE is not valid, the rest of the PDE is not defined.
- Multi-level page tables have some obvious advantages over approaches we've seen thus far. First, and perhaps most obviously, the multi-level table only allocates page-table space in proportion to the amount of address space you are using; thus it is generally compact and supports sparse address spaces. Second, if carefully constructed, each portion of the page table fits neatly within a page, making it easier to manage memory; the OS can simply grab the next free page when it needs to allocate or grow a page table.

Lecture Notes:

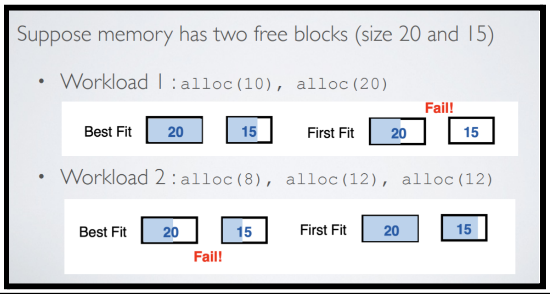
- **Managing Free Memory:**
- There are 2 types of memory allocation:
  1. **Static Allocation/Stack Allocation:**
     - Fixed in size.
     - Uses data structures that do not need to grow or shrink such as global and local variables.
       E.g. **char name[16];**
     - Done at compile time.
     - Restricted, but simple and efficient.
  2. **Dynamic Allocation/Heap Allocation:**
     - Changes in size.
     - Uses data structures that might increase/decrease in size according to different demands.
       E.g. **name = (char *) malloc(16);**
     - Done at run time.
     - General, but difficult to implement.
- Heap allocation is used to manage contiguous ranges of logical addresses.
- **malloc(size)** returns a pointer to a block of memory of at least size bytes, or NULL.
- **free(ptr)** releases the previously- allocated block pointed to by ptr.
- Heap allocation is difficult because using free() creates a lot of holes (**fragmentation**).
- Fragmentation is the inability to use memory that is free.
- Two factors are required for fragmentation:
  1. **Different lifetimes:** If all objects die at the same time, then there is no fragmentation.
  2. **Different sizes:** If all requests are the same size, then there is no fragmentation.
- Some important decisions:
  1. **Placement choice: Where in free memory to put a requested block?**
     - Freedom: Can select any memory in the heap.
     - Ideal: Put the block where it won't cause fragmentation later. This is impossible in general because it requires future knowledge.
  2. **Split free blocks to satisfy smaller requests?**
     - Freedom: Can choose any larger block to split.
     - Ideal: Choose specific blocks to minimize fragmentation.
- **Note:** Fragmentation is impossible to solve.
- Theoretical result: For any allocation algorithm, there exist streams of allocation and deallocation requests that defeat the allocator and force it into severe fragmentation L.
- **Heap Memory Allocator:**
- What the memory allocator must do:
  - Track which parts of memory are in use, and which parts are free. Ideally, there should be no wasted space and no time overhead.
- What the memory allocator cannot do:
  - Control the order of the number and size of requested blocks.
  - Know the number, size, & lifetime of future allocations.
- What makes a good memory allocator:
  - The one that avoids compaction (time consuming).
  - The one that minimizes fragmentation.
- **Tracking memory allocation with bitmaps:**
- **Bitmap:** 1 bit per allocation unit.
  0 means free
  1 means allocated

- Allocating a N-unit chunk requires scanning a bitmap for a sequence of N zero's.
- This process is very slow.
- **Tracking memory allocation with lists:**
- The **free lists** maintain a linked list of allocated and free segments.
- In an **implicit list**, each block has a header that records size and status (allocated or free). Searching for free blocks is linear in total number of blocks.
- An **explicit list** stores pointers in free blocks to create a doubly-linked list.
- **Freeing blocks:**
- Adjacent free blocks can be coalesced (merged).
  E.g.



- **Placement Algorithms:**
- There are 5 placement algorithms that can be used for merging free blocks:
  1. **First-fit:** Choose the first block that is large enough. The search can start at the beginning, or where the previous search ended.
  2. **Best-fit:** Choose the block that is closest in size to the request.
  3. **Worst-fit:** Choose the largest block.
  4. **Quick-fit:** Keep multiple free lists for common block sizes.
  5. **Buddy systems:** Round up allocations to power of 2 to make management faster.
- **Best Fit:**
- Minimizes fragmentation by allocating space from blocks that leave the smallest fragment.
- The best data structure to use is a heap as it is a list of free blocks where each has a header holding block size and a pointer to the next block.
- The idea is to search freelist for the block closest in size to the request.
- **First Fit:**
- Pick the first block that fits.
- Data structures that can be used for this include free list, sorted LIFO, FIFO, or by address.
- The idea is to scan the list and take the first one.

- **Best Fit vs First Fit:**

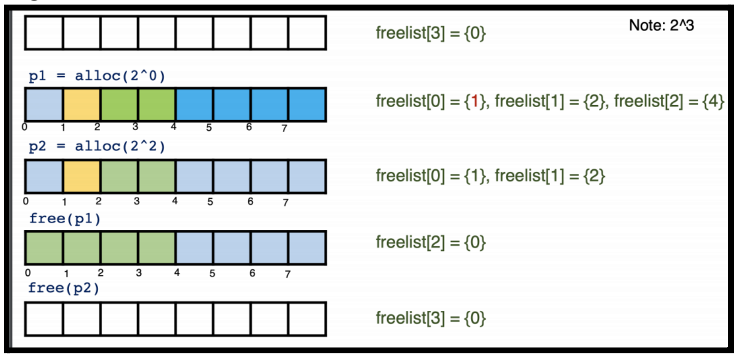


Suppose memory has two free blocks (size 20 and 15)

- Workload 1 : `alloc(10), alloc(20)`
- Workload 2 : `alloc(8), alloc(12), alloc(12)`

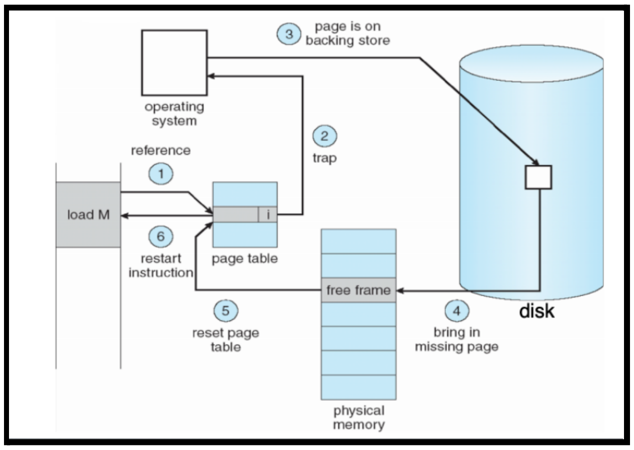
	First Fit	Best Fit
Advantage	Simplest, and often fastest and most efficient	In practice, similar storage utilization to first-fit
Disadvantage	May leave many small fragments near start of memory that must be searched repeatedly	Left-over fragments tend to be small (unusable)

- **Buddy Allocation:**
- Allocate blocks in 2^k.
- For the data structure, maintain n free lists of blocks of size $2^0, 2^1, \ldots, 2^n$.
- The idea is this:
    - Recursively divide larger blocks until they reach a suitable block.
    - Insert buddy blocks into free lists.
    - Upon free, recursively coalesce block with buddy if buddy is free.
      Note: The addresses of the buddy pair only differ by one bit.
- Linux uses this approach.
- E.g.

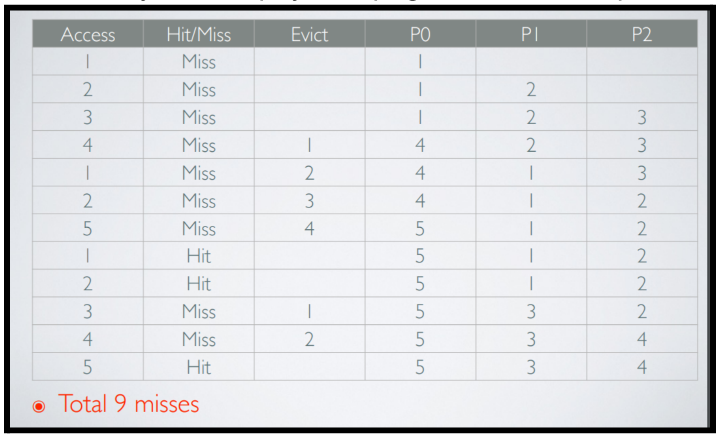


- Advantages:
    - Fast search (allocate) and merge (free).
    - Avoid iterating through the free list.
    - Avoid external fragmentation for req of 2^n.
    - Keep physical pages contiguous.

- **Page Replacements Algorithms:**
- Swapping is when we use a disk to simulate a larger virtual than physical memory. I.e.



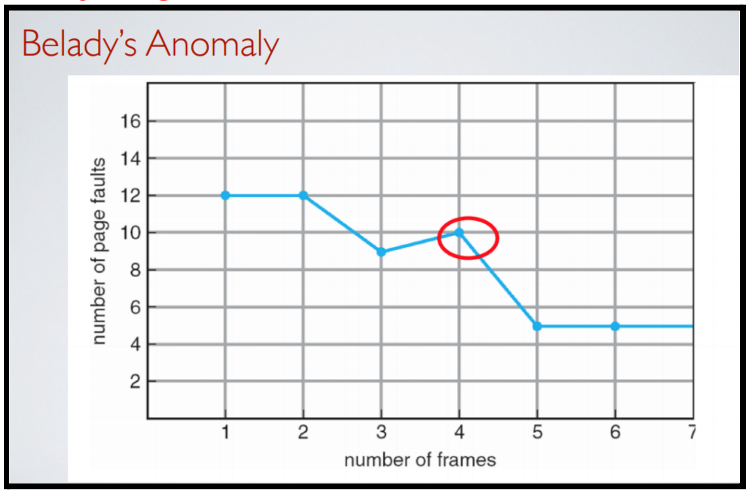
- What happens when there is a page fault? ➡ The OS loads the faulted page frame from disk into physical memory.
- What when there is no physical memory available or the process has reached its limit of maximum page frame allowed? ➡ The OS must evict an existing frame to replace it with the new one.
- How to determine which page frame should be evicted? ➡ The **page replacement algorithm**/**page eviction policy** determines which page frame to evict to minimize the fault rate (affecting paging performances).
- **Page Replacement Algorithms:**
- The goal of the replacement algorithm is to reduce the fault rate by selecting the best victim page to remove.
- There are 3 main algorithms:
    1. **FIFO (First In, First Out):** Evict the oldest page in the system.
    2. **LRU (Last Recently Used):** Evict the page that has not been used for the longest time in the past.
    3. **Second Chance:** An approximation of LRU that is more implementable.
- Replacement algorithms are evaluated on a reference string by counting the number of page faults.
- **FIFO:**
- Evict the oldest page in the system.
- We will only have 3 physical pages in this example.



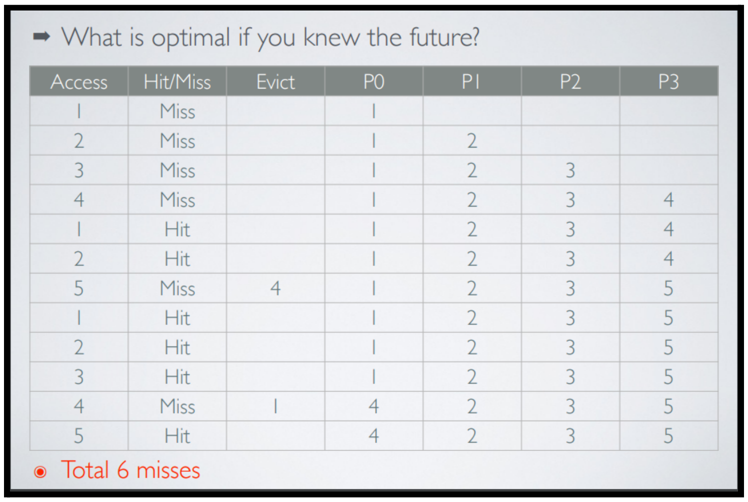
Access	Hit/Miss	Evict	P0	P1	P2
1	Miss		1		
2	Miss		1	2	
3	Miss		1	2	3
4	Miss	1	4	2	3
1	Miss	2	4	1	3
2	Miss	3	4	1	2
5	Miss	4	5	1	2
1	Hit		5	1	2
2	Hit		5	1	2
3	Miss	1	5	3	2
4	Miss	2	5	3	4
5	Hit		5	3	4

- Total 9 misses

- Does having more physical memory automatically means fewer page faults? The answer is no, more physical memory doesn't always mean fewer faults. This is proven by **Belady's Algorithm**.



- Belady tried to find the most optimal number of page frames if you could see the future, shown below.



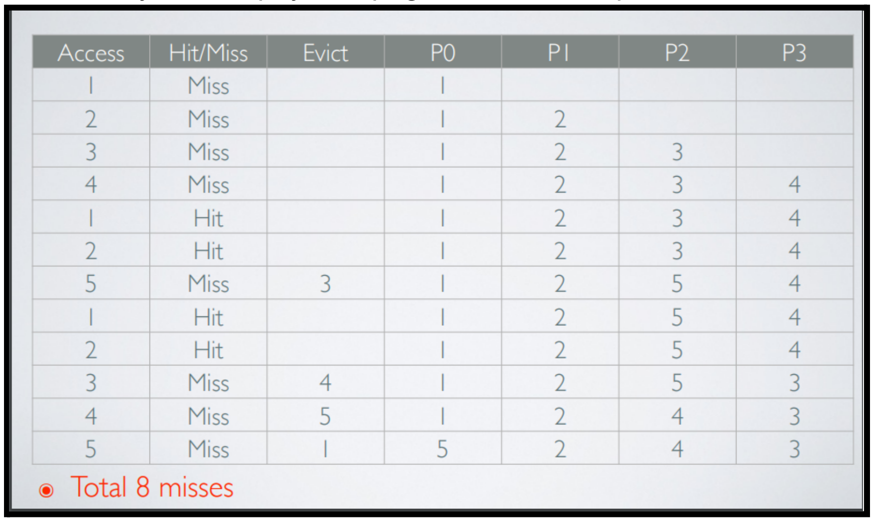
➡ What is optimal if you knew the future?

Access	Hit/Miss	Evict	P0	P1	P2	P3
1	Miss		1			
2	Miss		1	2		
3	Miss		1	2	3	
4	Miss		1	2	3	4
1	Hit		1	2	3	4
2	Hit		1	2	3	4
5	Miss	4	1	2	3	5
1	Hit		1	2	3	5
2	Hit		1	2	3	5
3	Hit		1	2	3	5
4	Miss	1	4	2	3	5
5	Hit		4	2	3	5

◉ Total 6 misses

- Belady's Algorithm is known and proven to be the optimal page replacement algorithm. The problem is that it is hard (nearly impossible) to predict the future.
  Belady's algorithm is useful to compare page replacement algorithms with the optimal to gauge room for improvement.

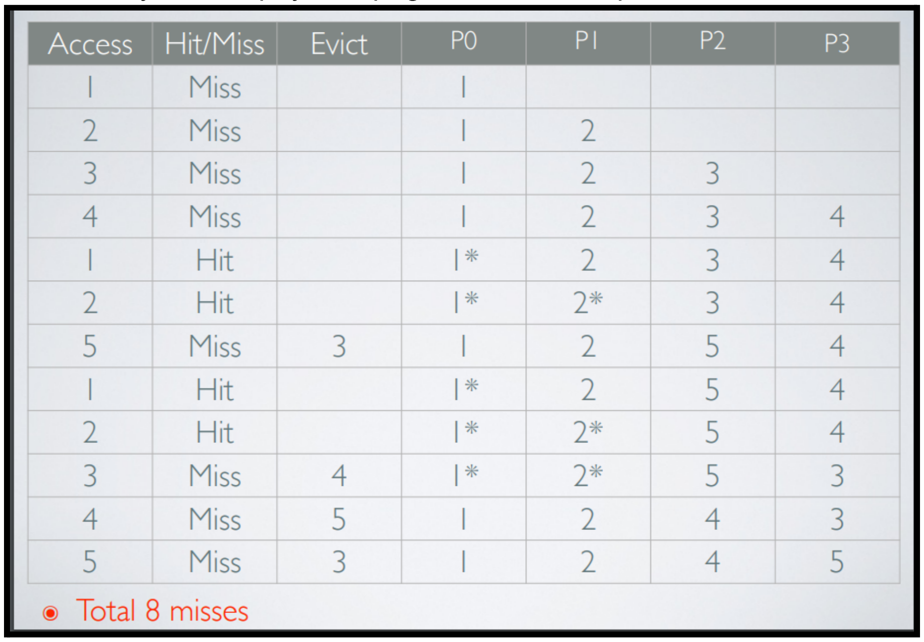
- **LRU:**
- Evict the page that has not been used for the longest time in the past.
- We will only have 3 physical pages in this example.



Access	Hit/Miss	Evict	P0	P1	P2	P3
1	Miss		1			
2	Miss		1	2		
3	Miss		1	2	3	
4	Miss		1	2	3	4
1	Hit		1	2	3	4
2	Hit		1	2	3	4
5	Miss	3	1	2	5	4
1	Hit		1	2	5	4
2	Hit		1	2	5	4
3	Miss	4	1	2	5	3
4	Miss	5	1	2	4	3
5	Miss	1	5	2	4	3

● Total 8 misses

- There are 2 ways to implement LRU:
  1. Stamp the pages with timer value.
     On access, stamp the PTE with the timer value.
     On a miss, scan the page table to find the oldest counter value.
     The problem is that this would double memory traffic.
  2. Use a doubly-linked list of pages.
     On access, move the page to the tail.
     On a miss, remove the head page.
     The problem with this is that, again, it is very expensive.
- So, we need to approximate LRU instead. This is where the Second Chance page replacement algorithm comes in.
- **Second Chance:**
- We will only have 3 physical pages in this example.



Access	Hit/Miss	Evict	P0	P1	P2	P3
1	Miss		1			
2	Miss		1	2		
3	Miss		1	2	3	
4	Miss		1	2	3	4
1	Hit		1*	2	3	4
2	Hit		1*	2*	3	4
5	Miss	3	1	2	5	4
1	Hit		1*	2	5	4
2	Hit		1*	2*	5	4
3	Miss	4	1*	2*	5	3
4	Miss	5	1	2	4	3
5	Miss	3	1	2	4	5

● Total 8 misses

- There are 2 ways to implement second chance:
    1. FIFO-like algorithm:
        - Use the accessed bit supported by most hardware.
        - Data structure: A linked list of pages with two pointers head and tail.
        - Code:
            - On hit, set the corresponding page's accessed bit to 1.
            - On miss:
                1. While the head's accessed bit is 1, set head's accessed bit to 0 and move it to tail.
                2. Otherwise, the head's accessed bit is 0, swap the head and move the new page to tail.
        - Good performances but requires moving pages on every miss.
    2. Clock algorithm:
        - Use the accessed bit supported by most hardware.
        - Data structure: A circular linked list of pages (clock) with 1 pointer (hand).
        - Code:
            - On hit, set the corresponding page's accessed bit to 1.
            - On miss:
                1. While the hand's accessed bit is 1, set the hand's accessed bit to 0 and move to the next page.
                2. Otherwise, if the hand's accessed bit is 0, swap the hand's page with the new page and move to the next page.
        - Better performances than fifo-like second chance (no rotation on miss)
- Some other replacement algorithms include:
    - **Random eviction:**
        - Very simple to implement.
        - Not overly horrible (avoids Belady's anomaly).
    - **LFU (least frequently used) eviction:**
        - Instead of just a bit, count the number of times each page was accessed. The least frequently accessed must not be very useful or maybe was just brought in and is about to be used.
        - Decay usage counts over time for pages that fall out of usage.
    - **MFU (most frequently used) algorithm:**
        - Because the page with the smallest count was probably just brought in and has yet to be used, it will be needed in the future.
- Neither LFU nor MFU are used very commonly.
- **Working Set Model:**
- How can we determine how much memory to give to each process?
- **Fixed space algorithms:**
    - Each process is given a limit of pages it can use.
    - When it reaches the limit, it replaces from its own pages.
    - Local replacement : Some processes may do well while others suffer.
- **Variable space algorithms:**
    - Process' set of pages grows and shrinks dynamically.
    - Global replacement: One process can ruin it for the rest.
- A **working set (WS)** of a process is used to model the dynamic locality of its memory usage.
- WS(t,w) = {pages P | P was referenced in the time interval (t, t-w)}
  t – time, w – working set window (measured in page refs)
- A page is in the working set only if it was referenced in the last w references.

- The **working set size** is the number of unique pages in the working set.
  I.e. It is the number of pages referenced in the interval (t, t-w).
- The working set size changes with program locality.
  During periods of poor locality, you reference more pages.
  Within that period of time, the working set size is larger.
- Intuitively, we want the working set to be the set of pages a process needs in memory to prevent heavy faulting.
  Each process has a parameter w that determines a working set with few faults.
  Don't run a process unless the working set is in memory.
- Some problems for the working set:
  1. Hard to determine w.
  2. Hard to know when the working set changes.
  - However, it is still used as an abstraction.
    For example, when people ask, "How much memory does Firefox need?", they are in effect asking for the size of Firefox's working set.
- **Page Fault Frequency (PFF):**
- **Page Fault Frequency (PFF)** is a variable space algorithm that uses a more ad-hoc approach.
- Monitor the fault rate for each process:
  - If the fault rate is above a high threshold, give it more memory.
  - If the fault rate is below a low threshold, take away memory.
- It is hard to use PFF to distinguish between changes in locality and changes in size of the working set.
- **Thrashing:**
- An **overcommitted system** occurs when an OS spends most of the time paging data back and forth from the disk and spending little time doing useful work.
- The problem comes from either:
  - A bad page replacement algorithm (that does not help minimizing page fault) OR
  - There is not enough physical memory for all processes.
- **Windows XP Paging Policy:**
- Local page replacement.
- Per-process FIFO.
- Processes start with a default of 50 pages.
- XP monitors page fault rate and adjusts working-set size accordingly.
- On page faults, clusters of pages around the missing page are brought into memory.
- **Linux Paging:**
- Global replacement (like most Unix).
- Modified second-chance clock algorithm.
- Pages age with each pass of the clock hand.
- Pages that are not used for a long time will eventually have a value of zero.

<u>**Textbook Notes:**</u>

- **Beyond Physical Memory - Mechanisms:**
- <u>Introduction:</u>
- Thus far, we've assumed that an address space is unrealistically small and fits into physical memory. In fact, we've been assuming that every address space of every running process fits into memory. We will now relax these big assumptions, and assume that we wish to support many concurrently-running large address spaces.
- To do so, we require an additional level in the memory hierarchy. Thus far, we have assumed that all pages reside in physical memory. However, to support large address spaces, the OS will need a place to stash away portions of address spaces that currently aren't in great demand. In general, the characteristics of such a location are that it

should have more capacity than memory; as a result, it is generally slower. In modern systems, this role is usually served by a hard disk drive. Thus, in our memory hierarchy, big and slow hard drives sit at the bottom, with memory just above. This is for convenience and ease of use.

- With a large address space, you don't have to worry about if there is room enough in memory for your program's data structures; rather, you just write the program naturally, allocating memory as needed. It is a powerful illusion that the OS provides, and makes your life vastly simpler. A contrast is found in older systems that used **memory overlays**, which required programmers to manually move pieces of code or data in and out of memory as they were needed.
- Beyond just a single process, the addition of swap space allows the OS to support the illusion of a large virtual memory for multiple concurrently running processes. The invention of multiprogramming almost demanded the ability to swap out some pages, as early machines clearly could not hold all the pages needed by all processes at once. Thus, the combination of multiprogramming and ease-of-use leads us to want to support using more memory than is physically available. It is something that all modern VM systems do.
- Swap Space:
- The first thing we will need to do is to reserve some space on the disk for moving pages back and forth. In operating systems, we generally refer to such space as **swap space**, because we swap pages out of memory to it and swap pages into memory from it. Thus, we will simply assume that the OS can read from and write to the swap space, in page-sized units. To do so, the OS will need to remember the **disk address** of a given page.
- The size of the swap space is important, as ultimately it determines the maximum number of memory pages that can be in use by a system at a given time.
- We should note that swap space is not the only on-disk location for swapping traffic. For example, assume you are running a program binary. The code pages from this binary are initially found on disk, and when the program runs, they are loaded into memory (either all at once when the program starts execution, or, as in modern systems, one page at a time when needed). However, if the system needs to make room in physical memory for other needs, it can safely re-use the memory space for these code pages, knowing that it can later swap them in again from the on-disk binary in the file system.
- The Present Bit:
- Let us assume, for simplicity, that we have a system with a hardware-managed TLB.
- Recall first what happens on a memory reference. The running process generates virtual memory references (for instruction fetches, or data accesses), and, in this case, the hardware translates them into physical addresses before fetching the desired data from memory.
- Remember that the hardware first extracts the VPN from the virtual address, checks the TLB for a match (a TLB hit), and if a hit, produces the resulting physical address and fetches it from memory. This is hopefully the common case, as it is fast (requiring no additional memory accesses). If the VPN is not found in the TLB (a TLB miss), the hardware locates the page table in memory using the page table base register and looks up the page table entry (PTE) for this page using the VPN as an index. If the page is valid and present in physical memory, the hardware extracts the PFN from the PTE, installs it in the TLB, and retries the instruction, this time generating a TLB hit.
- If we wish to allow pages to be swapped to disk, however, we must add even more machinery. Specifically, when the hardware looks in the PTE, it may find that the page is not present in physical memory. The way the hardware or OS determines this is through a new piece of information in each page-table entry, known as the **present bit**. If the

present bit is set to one, it means the page is present in physical memory and everything proceeds as above; if it is set to zero, the page is not in memory but rather on disk somewhere. The act of accessing a page that is not in physical memory is commonly referred to as a **page fault**.

- Upon a page fault, the OS is invoked to service the page fault. A particular piece of code, known as a **page-fault handler**, runs, and must service the page fault.
- The Page Fault:
- Recall that with TLB misses, we have two types of systems: hardware-managed TLBs (where the hardware looks in the page table to find the desired translation) and software-managed TLBs (what the OS does). In either type of system, if a page is not present, the OS is put in charge to handle the page fault. The **OS page-fault handler** runs to determine what to do. Virtually all systems handle page faults in software. Even with a hardware-managed TLB, the OS manages this important duty.
- If a page is not present and has been swapped to disk, the OS will need to swap the page into memory in order to service the page fault. Thus, a question arises: how will the OS know where to find the desired page? In many systems, the page table is a natural place to store such information. Thus, the OS could use the bits in the PTE normally used for data such as the PFN of the page for a disk address. When the OS receives a page fault for a page, it looks in the PTE to find the address, and issues the request to disk to fetch the page into memory.
- When the disk I/O completes, the OS will then update the page table to mark the page as present, update the PFN field of the page-table entry (PTE) to record the in-memory location of the newly-fetched page, and retry the instruction. This next attempt may generate a TLB miss, which would then be serviced and update the TLB with the translation (one could alternately update the TLB when servicing the page fault to avoid this step). Finally, a last restart would find the translation in the TLB and thus proceed to fetch the desired data or instruction from memory at the translated physical address.
- Note that while the I/O is in flight, the process will be in the blocked state. Thus, the OS will be free to run other ready processes while the page fault is being serviced. Because I/O is expensive, this overlap of the I/O (page fault) of one process and the execution of another is yet another way a multiprogrammed system can make the most effective use of its hardware.
- What If Memory Is Full:
- In the process described above, you may notice that we assumed there is plenty of free memory in which to page in a page from swap space. Of course, this may not be the case. Thus, the OS might like to first page out one or more pages to make room for the new page(s) the OS is about to bring in. The process of picking a page to kick out, or replace is known as the **page-replacement policy**.
- As it turns out, a lot of thought has been put into creating a good page replacement policy, as kicking out the wrong page can exact a great cost on program performance. Making the wrong decision can cause a program to run at disk-like speeds instead of memory-like speeds; in current technology that means a program could run 10,000 or 100,000 times slower.
- Page Fault Control Flow:
- There are now three important cases to understand when a TLB miss occurs.
- First, that the page was both present and valid. In this case, the TLB miss handler can simply grab the PFN from the PTE, retry the instruction this time resulting in a TLB hit, and thus continue as described before.
- In the second case, the page fault handler must be run. Although this was a legitimate page for the process to access it is not present in physical memory.

- Third, the access could be to an invalid page, due for example to a bug in the program. In this case, no other bits in the PTE really matter; the hardware traps this invalid access, and the OS trap handler runs, likely terminating the offending process.
- This is what the OS roughly must do in order to service the page fault. First, the OS must find a physical frame for the soon-to-be-faulted-in page to reside within; if there is no such page, we'll have to wait for the replacement algorithm to run and kick some pages out of memory, thus freeing them for use here. With a physical frame in hand, the handler then issues the I/O request to read in the page from swap space. Finally, when that slow operation completes, the OS updates the page table and retries the instruction. The retry will result in a TLB miss, and then, upon another retry, a TLB hit, at which point the hardware will be able to access the desired item.
- When Replacements Really Occur:
- Thus far, the way we've described how replacements occur assumes that the OS waits until memory is entirely full, and only then replaces a page to make room for some other page. As you can imagine, this is a little bit unrealistic, and there are many reasons for the OS to keep a small portion of memory free more proactively.
- To keep a small amount of memory free, most operating systems thus have some kind of high watermark (HW) and low watermark (LW) to help decide when to start evicting pages from memory. How this works is as follows: when the OS notices that there are fewer than LW pages available, a background thread that is responsible for freeing memory runs. The thread evicts pages until there are HW pages available. The background thread, sometimes called the **swap daemon** or **page daemon**, then goes to sleep, happy that it has freed some memory for running processes and the OS to use.
- Beyond Physical Memory - Policies:
- In a virtual memory manager, life is easy when you have a lot of free memory. A page fault occurs, you find a free page on the free-page list, and assign it to the faulting page. Unfortunately, things get a little more interesting when little memory is free. In such a case, this memory pressure forces the OS to start paging out pages to make room for actively-used pages. Deciding which page(s) to evict is encapsulated within the replacement policy of the OS. Historically, it was one of the most important decisions the early virtual memory systems made, as older systems had little physical memory.
- Cache Management:
- Given that main memory holds some subset of all the pages in the system, it can rightly be viewed as a cache for virtual memory pages in the system. Thus, our goal in picking a replacement policy for this cache is to minimize the number of cache misses, i.e., to minimize the number of times that we have to fetch a page from disk. Alternatively, one can view our goal as maximizing the number of cache hits, i.e., the number of times a page that is accessed is found in memory.
- Knowing the number of cache hits and misses let us calculate the **average memory access time (AMAT)** for a program.

$$\boxed{AMAT = T_M + (P_{Miss} \cdot T_D)}$$

T_M represents the cost of accessing memory, T_D the cost of accessing disk, and P_{Miss} the probability of not finding the data in the cache. P_{Miss} varies from 0.0 to 1.0, and sometimes we refer to a percent miss rate instead of a probability (E.g. A 10% miss rate means PMiss = 0.10).

Note: You always pay the cost of accessing the data in memory. When you miss, however, you must additionally pay the cost of fetching the data from disk.

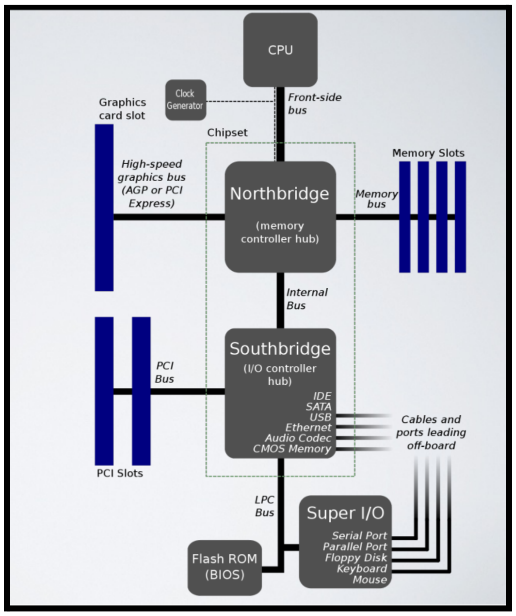
- The Optimal Replacement Policy:
- The logic is like this: If you have to throw out some page, why not throw out the one that is needed the furthest from now? By doing so, you are essentially saying that all the other pages in the cache are more important than the one furthest out. The reason this is true is simple: you will refer to the other pages before you refer to the one furthest out.
- In the computer architecture world, architects sometimes find it useful to characterize misses by type, into one of three categories: **compulsory**, **capacity**, and **conflict misses**, sometimes called the Three C's.
- A **compulsory miss** or **cold-start miss** occurs because the cache is empty to begin with and this is the first reference to the item.
- A **capacity miss** occurs because the cache ran out of space and had to evict an item to bring a new item into the cache.
- A **conflict miss** arises in hardware because of limits on where an item can be placed in a hardware cache, due to **set associativity**. It does not arise in the OS page cache because such caches are always fully-associative. I.e. There are no restrictions on where in memory a page can be placed.
- A Simple Policy - FIFO:
- Many early systems avoided the complexity of trying to approach optimal and employed very simple replacement policies.
- For example, some systems used **FIFO replacement**, where pages were simply placed in a queue when they enter the system. When a replacement occurs, the page on the tail of the queue is evicted. FIFO has one great strength: it is quite simple to implement.
- Another Simple Policy - Random:
- Another similar replacement policy is Random, which simply picks a random page to replace under memory pressure. Random has properties similar to FIFO; it is simple to implement, but it doesn't really try to be too intelligent in picking which blocks to evict.
- Using History - LRU:
- Unfortunately, any policy as simple as FIFO or Random is likely to have a common problem: it might kick out an important page, one that is about to be referenced again. FIFO kicks out the page that was first brought in; if this happens to be a page with important code or data structures upon it, it gets thrown out anyhow, even though it will soon be paged back in.
- As we did with scheduling policy, to improve our guess at the future, we once again lean on the past and use history as our guide.
- One type of historical information a page-replacement policy could use is **frequency**; if a page has been accessed many times, perhaps it should not be replaced as it clearly has some value.
- A more commonly used property of a page is its **recency of access**; the more recently a page has been accessed, perhaps the more likely it will be accessed again.
- This family of policies is based on what people refer to as the **principle of locality**, which basically is just an observation about programs and their behavior. What this principle says, quite simply, is that programs tend to access certain code sequences (e.g. in a loop) and data structures (e.g. an array accessed by the loop) quite frequently; we should thus try to use history to figure out which pages are important, and keep those pages in memory when it comes to eviction time.
- And thus, a family of simple historically-based algorithms are born. The **Least-Frequently-Used (LFU)** policy replaces the least-frequently used page when an eviction must take place. Similarly, the **Least-Recently-Used (LRU)** policy replaces the least-recently-used page.

- There are two types of locality that programs tend to exhibit. The first is known as **spatial locality**, which states that if a page P is accessed, it is likely the pages around it (say P − 1 or P + 1) will also likely be accessed. The second is **temporal locality**, which states that pages that have been accessed in the near past are likely to be accessed again in the near future. The assumption of the presence of these types of locality plays a large role in the caching hierarchies of hardware systems, which deploy many levels of instruction, data, and address-translation caching to help programs run fast when such locality exists.
- Of course, the **principle of locality**, as it is often called, states that there is no hard-and-fast rule that all programs must obey. Indeed, some programs access memory or disk in rather random fashion and don't exhibit much or any locality in their access streams. Thus, while locality is a good thing to keep in mind while designing caches of any kind (hardware or software), it does not guarantee success. Rather, it is a heuristic that often proves useful in the design of computer systems.
- Approximating LRU:
- The idea requires some hardware support, in the form of a **use bit**, sometimes called the **reference bit**, the first of which was implemented in the first system with paging.
- There is one use bit per page of the system, and the use bits live in memory somewhere. Whenever a page is referenced (read or written), the use bit is set by hardware to 1. The hardware never clears the bit. That is the responsibility of the OS.
- How does the OS employ the use bit to approximate LRU? Well, there could be a lot of ways, but with the clock algorithm, one simple approach was suggested. Imagine all the pages of the system arranged in a circular list. A clock hand points to some particular page to begin with (it doesn't really matter which). When a replacement must occur, the OS checks if the currently-pointed page P has a use bit of 1 or 0. If 1, this implies that page P was recently used and thus is not a good candidate for replacement. Thus, the use bit for P is set to 0 (cleared), and the clock hand is incremented to the next page (P + 1). The algorithm continues until it finds a use bit that is set to 0, implying this page has not been recently used (or, in the worst case, that all pages have been and that we have now searched through the entire set of pages, clearing all the bits).
- Note that this approach is not the only way to employ a use bit to approximate LRU. Indeed, any approach which periodically clears the use bits and then differentiates between which pages have use bits of 1 versus 0 to decide which to replace would be fine.
- Considering Dirty Pages:
- One small modification to the clock algorithm that is commonly made is the additional consideration of whether a page has been modified or not while in memory. The reason for this: if a page has been modified and is thus dirty, it must be written back to disk to evict it, which is expensive. If it has not been modified and is thus clean, the eviction is free; the physical frame can simply be reused for other purposes without additional I/O. Thus, some VM systems prefer to evict clean pages over dirty pages.
- To support this behavior, the hardware should include a **modified bit**/**dirty bit**. This bit is set any time a page is written, and thus can be incorporated into the page-replacement algorithm. The clock algorithm, for example, could be changed to scan for pages that are both unused and clean to evict first; failing to find those, then for unused pages that are dirty, and so forth.
- Other VM Policies:
- Page replacement is not the only policy the VM subsystem employs though it may be the most important. For example, the OS also has to decide when to bring a page into memory. This policy, sometimes called the **page selection policy**, presents the OS with some different options.

- For most pages, the OS simply uses **demand paging**, which means the OS brings the page into memory when it is accessed, “on demand” as it were. Of course, the OS could guess that a page is about to be used, and thus bring it in ahead of time; this behavior is known as **prefetching** and should only be done when there is a reasonable chance of success. For example, some systems will assume that if a code page P is brought into memory, that code page P + 1 will likely soon be accessed and thus should be brought into memory too.
- Another policy determines how the OS writes pages out to disk. Of course, they could simply be written out one at a time; however, many systems instead collect a number of pending writes together in memory and write them to disk in one (more efficient) write. This behavior is usually called **clustering** or simply grouping of writes, and is effective because of the nature of disk drives, which perform a single large write more efficiently than many small ones.
- Thrashing:
- Before closing, we address one final question: what should the OS do when memory is simply oversubscribed, and the memory demands of the set of running processes simply exceeds the available physical memory? In this case, the system will constantly be paging, a condition sometimes referred to as **thrashing**.
- Some earlier operating systems had a fairly sophisticated set of mechanisms to both detect and cope with thrashing when it took place. For example, given a set of processes, a system could decide not to run a subset of processes, with the hope that the reduced set of processes’ working sets (the pages that they are using actively) fit in memory and thus can make progress. This approach, generally known as **admission control**, states that it is sometimes better to do less work well than to try to do everything at once poorly, a situation we often encounter in real life as well as in modern computer systems.
- Some current systems take a more draconian approach to memory overload. For example, some versions of Linux run an out-of-memory killer when memory is oversubscribed; this daemon chooses a memory intensive process and kills it, thus reducing memory in a none-too-subtle manner. While successful at reducing memory pressure, this approach can have problems, if, for example, it kills the X server and thus renders any applications requiring the display unusable.

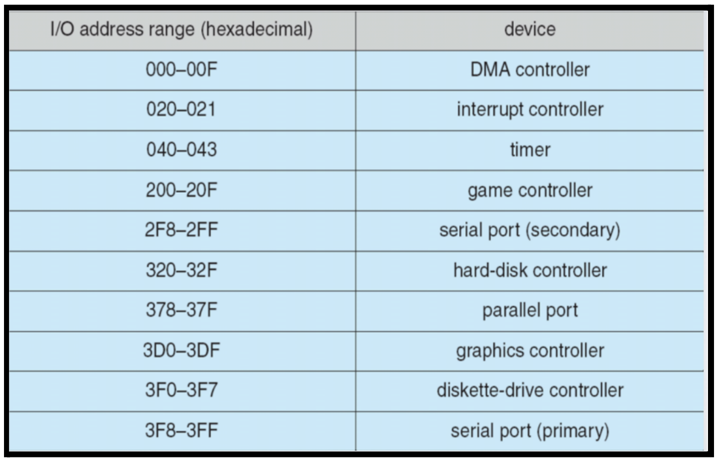
Lecture Notes:

- **I/O:**
- I/O devices vary greatly and new types of I/O devices appear frequently.
- There are various methods to control them and to manage their performances.
- Ports, buses, and device controllers connect to various devices.
- Some I/O Device interfaces are:
    - **Port:** A connection point for device.
      E.g. Serial Port
    - **Bus:** A daisy chain or shared direct access.
      E.g. Peripheral Component Interconnect Bus (PCI)
      E.g. Universal Serial Bus (USB)
    - **Controller (host adapter):** A set of electronics that operate ports, buses, devices, etc.
      It can be integrated or separated (host adapter).
      It contains processors, microcodes, private memory, bus controllers, etc.
      E.g. Northbridge, Southbridge, graphics controller, DMA, NIC, etc
- I/O Architecture:



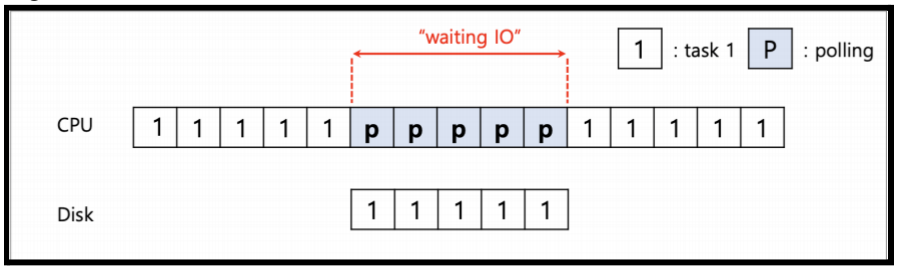
- Each device has three types of registers and the OS controls the device by reading or writing these registers.
  These registrars are:
    1. **Status register:** See the current status of the device.
    2. **Command register/Control register:** Tell the device to perform a certain task.
    3. **Data register:** Pass data to the device, or get data from the device.
- There are 2 ways to read/write with these registers:
    1. **I/O ports:** Uses in and out instructions on x86 to read and write devices registers.
    2. **Memory-mapped I/O:** Device registers are available as if they were memory locations and the OS can load (read) or store (write) to the device.

- Table of I/O Ports on PC:

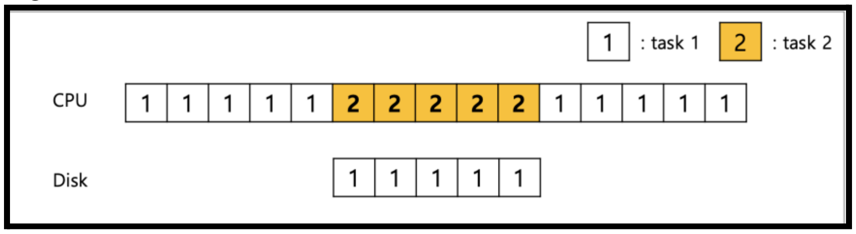


I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

- **Polling:**
- Here, the OS waits until the device is ready by repeatedly reading the status register.
- While it is simple and works, it wastes CPU time just waiting for the device.
- E.g.

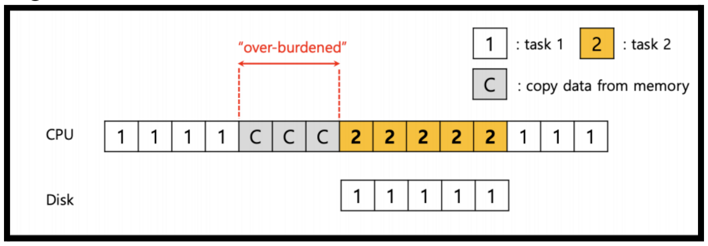


- **Interrupts:**
- With interrupts:
    1. Put the I/O request process to sleep and switch context.
    2. When the device is finished, send an interrupt to wake the process waiting for the I/O.
- The CPU is properly utilized.
- E.g.

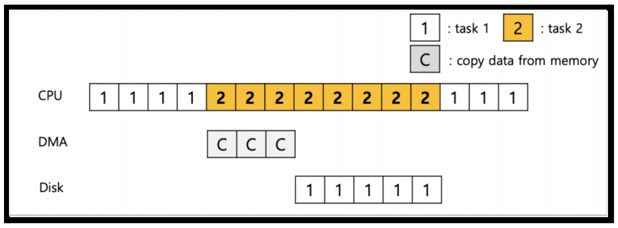


- **Polling vs Interrupts:**
- Interrupts are not always the best solution.
- If the device performs very quickly, interrupts will slow down the system.
- E.g. For a high network packet arrival rate:
    - Packets can arrive faster than the OS can process them.
    - Interrupts are very expensive (context switch).
    - Interrupt handlers have high priority.
    - In the worst case, it can spend 100% of time in the interrupt handler and never make any progress (receive livelock).
- The best practice is to do adaptive switching between interrupts and polling.

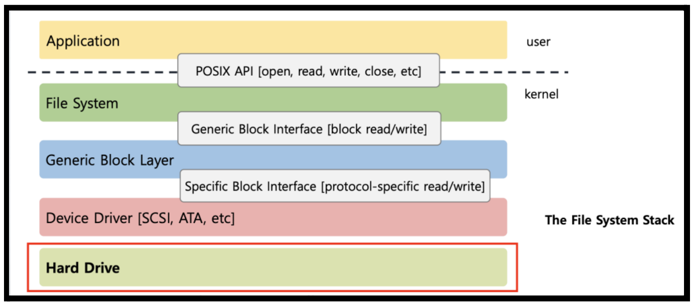
- **Data Copying:**
- When the OS copies data, the CPU wastes a lot of time in copying a large chunk of data from memory to the device.
- E.g.



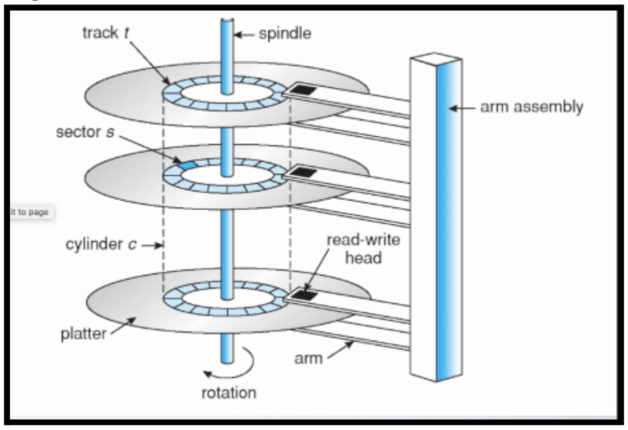
- As a result, we won't use interrupts or polling. We'll use **Direct Memory Access (DMA)**.
- **DMA (Direct Memory Access):**
- Only use the CPU to transfer control requests, not data, by passing buffer locations in memory.
- The device reads the list and accesses buffers through DMA.
- Descriptions sometimes allow for scatter/gather I/O.
- Here are the steps:
    1. OS writes DMA command block into memory.
    2. DMA bypasses CPU to transfer data directly between I/O device and memory.
    3. When completed, DMA raises an interrupt.
- E.g.



- **Note:** Variety is a challenge.
- The problem is that there are many devices and each has its own protocol.
    - Some devices are accessed by I/O ports or memory mapping or both.
    - Some devices can interact by polling or interrupt or both.
    - Some devices can transfer data by programmed I/O or DMA or both.
- The solution is to use an abstraction.
    - Build a common interface.
    - Write a device driver for each device.
- Drivers are 70% of the Linux source code.
- **File System Abstraction:**
- A file system specifics of which disk class it is using It issues block read and write requests to the generic block layer.
- I.e.


Application
user
POSIX API [open, read, write, close, etc]
kernel
File System
Generic Block Interface [block read/write]
Generic Block Layer
Specific Block Interface [protocol-specific read/write]
Device Driver [SCSI, ATA, etc]
The File System Stack
Hard Drive

- **Hard Disk Drive (HDD):**
- **Platter (aluminum coated with a thin magnetic layer):**
    - A circular hard surface.
    - Data is stored persistently by inducing magnetic changes to it.
    - Each platter has 2 sides, each of which is called a surface.
- **Spindle:**
    - Spindle is connected to a motor that spins the platters around.
    - The rate of rotations is measured in RPM (Rotations Per Minute).
    - Typical modern values are: 7,200 RPM to 15,000 RPM.
- **Track:**
    - Concentric circles of sectors.
    - Data is encoded on each surface in a track.
    - A single surface contains many thousands and thousands of tracks.
- **Cylinder:**
    - A stack of tracks of fixed radius.
    - Heads record and sense data along cylinders.
    - Generally only one head is active at a time.
- E.g.

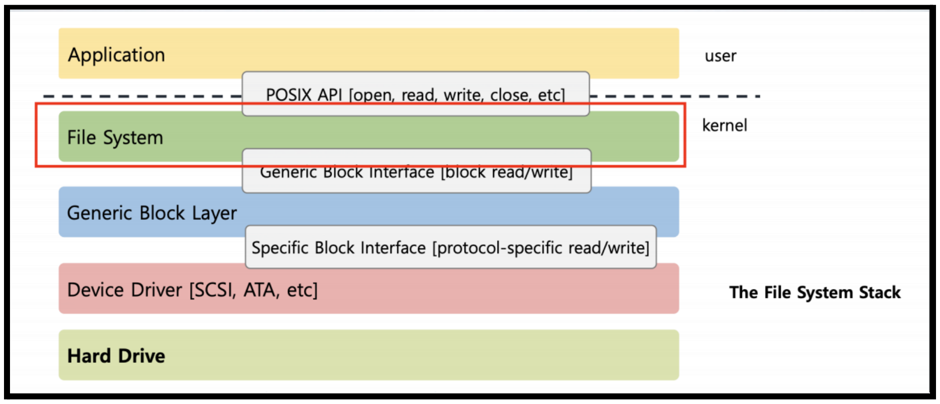


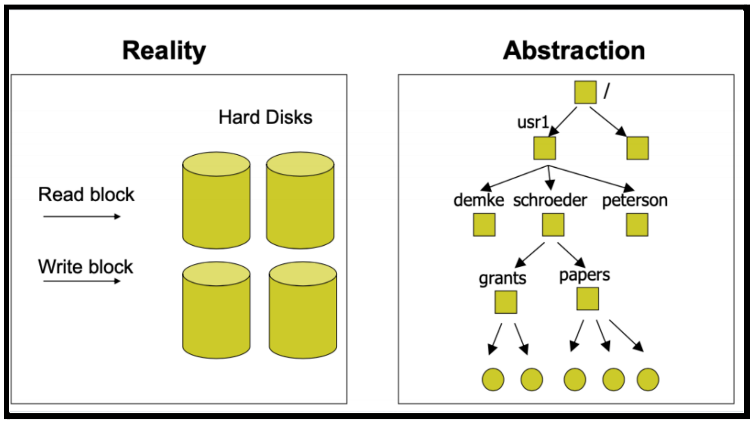
- The disk interface presents a linear array of sectors.
- Historically it is 512 Bytes but 4 KB in "advanced format" disks.
- Written atomically (even if there is a power failure).
- Disk maps logical sector numbers to physical sectors.
- The OS doesn't know the logical to physical sector mapping.
- Each time we write/read, we need to do 3 things:
    1. **Seek:**
        - Move the head to the above specific track.
            1. Speedup: Accelerate arm to max speed.
            2. Coast: At max speed (for long seeks).
            3. Slowdown: Stops arm near destination.
            4. Settle: Adjusts head to the actual desired track.
        - Seek is slow.
        - Settling alone can take 0.5 to 2ms.
        - An entire seek operation often takes 4 - 10ms.
    2. **Rotate:**
        - Rotate disk until the head is above the right sector.
        - This depends on rotations per minute (RPM).
        - With typical 7200 RPM it takes 8.3 ms / rotation.
        - The average rotation is slow (4.15 ms).

3. **Transfer:**
    - Data is either read from or written to the surface..
    - The speed depends on RPM and sector density.
    - With typical 100+ MB/s it takes 5μs / sector (512 bytes).
    - Pretty Fast.
- In summary:
    - Seeks are slow .
    - Rotations are slow.
    - Transfers are fast.
- What kind of workload is fastest for disks:
    - **Sequential:** Access sectors in order (transfer dominated).
    - **Random:** Access sectors arbitrarily (seek+rotation dominated).
- The Disk Scheduler decides which I/O request to schedule next:
    - First Come First Served (FCFS)
    - Shortest Seek Time First (SSTF)
    - Elevator Scheduling (SCAN)
- **Solid State Drive (SSD):**
- Completely solid state (there are no moving parts).
- It remembers data by storing charge (like RAM).
- Advantages:
    - Same interface as HDD (linear array of sectors).
    - No mechanical seek and rotation times to worry about. SSD are way faster than HDD.
    - Lower power consumption and heat (better for mobile devices).
- Disadvantages:
    - More expensive than HDD for now but it is getting cheaper.
    - Limited durability as charge wears out over time (but improving).
    - There is a limited number of overwrites possible.
      Blocks wear out after 10,000 (MLC) – 100,000 (SLC) erases.
      Requires **Flash Translation Layer (FTL)** to provide wear levelling, so repeated writes to logical block don't wear out physical block.
      FTL can seriously impact performance.

Lecture Notes:

- **File System:**
- The file system provides an abstraction.
- It specifies which disk class it is using.
- It issues block read/write requests to the generic block layer.
- I.e.


Application
user
POSIX API [open, read, write, close, etc]
File System
kernel
Generic Block Interface [block read/write]
Generic Block Layer
Specific Block Interface [protocol-specific read/write]
Device Driver [SCSI, ATA, etc]
The File System Stack
Hard Drive



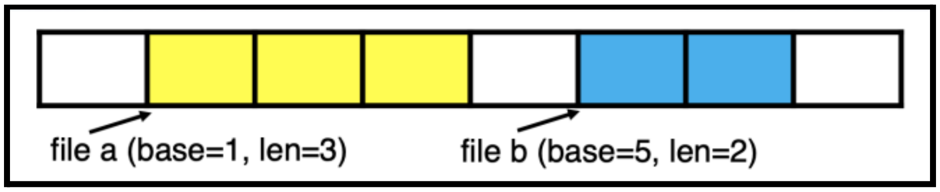
- The goals of a file system are:
    - Implement an abstraction (files) for secondary storage.
    - Organize files logically (directories).
    - Permit sharing of data between processes, people, and machines.
    - Protect data from unwanted access (security).
- **Files:**
- A **file** is named bytes on a disk that encapsulates data with some properties such as: contents, size, owner, last read/write time, protection, etc.
- A file can also have a type:
    - The type is understood by the file system: block device, character device, link, FIFO, socket, etc.
    - The type is also understood by other parts of the OS or runtime libraries: text, image, source, compiled libraries (Unix .so and Windows .dll), executable, etc.
- A file's type can be encoded in its name or contents:
    - Windows encodes type in name: .com, .exe, .bat, .dll, .jpg, etc.
    - Unix encodes type in contents: magic numbers, initial characters (E.g. #! for shell scripts).
- **Note:** In Unix, everything is a file.

- There are several methods for accessing files:
  1. **Sequential access:**
  - Used by file systems and is the most common method.
  - Read bytes one at a time, in order.
  2. **Random access:**
  - Used by file systems.
  - Random access is given to a block/byte number (read/write bytes at offset n).
  3. **Indexed access:**
  - Used by databases.
  - The file system contains an index to a particular field of each record in a file.
  - Reads specify a value for that field and the system finds the record via the index.
  4. **Record access:**
  - Used by databases.
  - The file is an array of fixed-or-variable-length records.
  - Read/write sequentially or randomly by record number.
- Some basic file operations for Unix & Windows are:

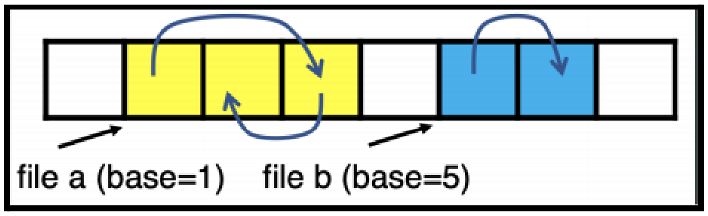
Unix	**Windows**
create(name)	CreateFile(name, CREATE)
open(name, how)	CreateFile(name, OPEN)
read(fd, buf, len)	ReadFile(handle, …)
write(fd, buf, len)	WriteFile(handle, …)
sync(fd)	FlushFileBuffers(handle, …)
seek(fd, pos)	SetFilePointer(handle, …)
close(fd)	CloseHandle(handle, …)
unlink(name)	DeleteFile(name)
	CopyFile(name)
	MoveFile(name)

- How to Track File's Data:
  - Disk management:
    - Need to keep track of where file contents are on disk.
    - Must be able to use this to map byte offset to disk block.
    - Structure tracking a file's blocks is called an index node or inode.
    - Inodes must be stored on disk, too.
  - Things to keep in mind while designing file structure:
    - Most files are small.
    - Much of the disk is allocated to large files.
    - Many of the I/O operations are made to large files.
    - Want good sequential and good random access.

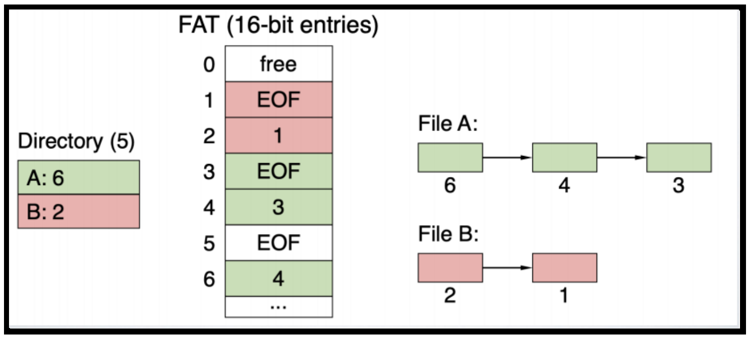
- **Straw Man #1 - Contiguous Allocation:**
- Is **extent-based**, meaning that it allocates files like segmented memory.
- When creating a file, make the user pre-specify its length and allocate all space at once.
- Inode contents: location and size.
- E.g.



- Advantage: Simple, fast access, both sequential and random.
- Disadvantage: External fragmentation (similar to VM).
- **Straw Man #2 - Linked Files:**
- Basically a linked list on disk:
    - Keep a linked list of all free blocks.
    - Inode contents: A pointer to file's first block.
    - In each block, keep a pointer to the next one.
- E.g.

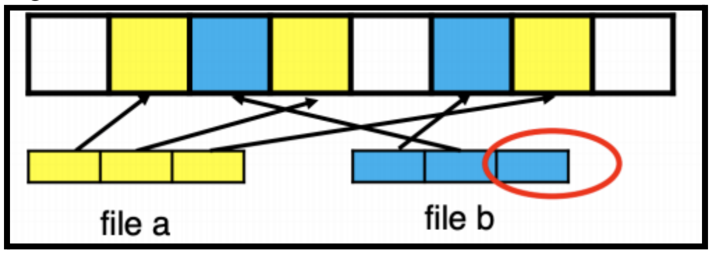


- Advantage: Easy dynamic growth & sequential access, no fragmentation.
- Disadvantage: Linked lists on disk are a bad idea because of access times. Random access is very slow (E.g. You have to traverse the whole file to find the last block). Pointers take up room in blocks, skewing alignment.
- **DOS FAT:**
- Linked files with key optimization: Puts links in fixed-size "file allocation table" (FAT) rather than in the block.
- However, this still does pointer chasing.
- E.g.



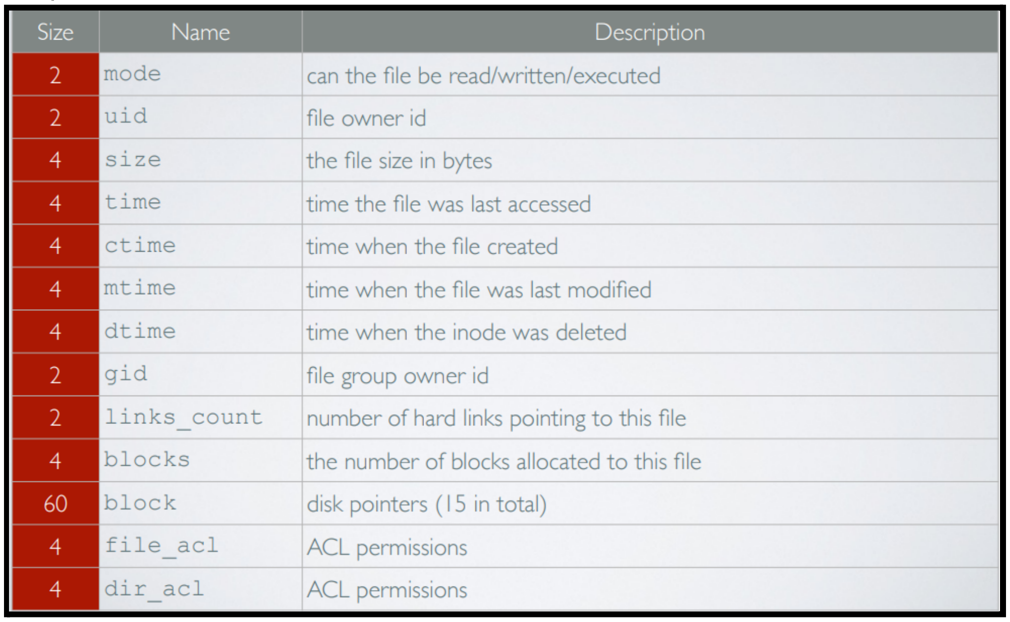
- Given entry size = 16 bits (initial FAT16 in MS-DOS 3.0), what's the maximum size of the FAT → 65,536
- Given a 512 byte block, what's the maximum size of FS → 32MB
- What is the space overhead → 2 bytes / 512 byte block = ~0.4%
- How to protect against errors → Create duplicate copies of FAT on disk (State duplication is a very common theme in reliability).
- Where is the root directory → Fixed location on disk.

- **Indexed Files:**
- Each file has a table holding all of its block pointers:
    - Max file size fixed by table's size.
    - Allocate table to hold file's block pointers on file creation.
    - Allocate actual blocks on demand using the free list.
- E.g.



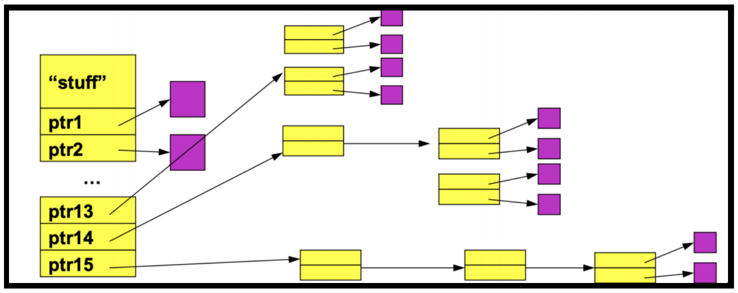
- Adv: Both sequential and random access are easy.
- Disadv: Mapping table requires a large chunk of contiguous space.
- **Unix File System:**
- The disk is (physically) divided into sectors (usually 512 bytes per sector).
- The file system is (logically) divided into blocks (E.g. 4 KB per block).
- Disk space is allocated in granularity of blocks:
    1. The data blocks "D" stored files (and directories) content.
    2. The inodes block "I" stores the inode table.
    3. The data bitmap "d" block d tracks which data block is free or allocated (one bit per block on the disk).
    4. The inode bitmap "i" block i tracks which inode is free or allocated (one bit per inode).
    5. The Superblock "S" (a.k.a Master Block or partition control block) contains:
        a. A magic number to identify the file system type.
        b. The number of blocks dedicated to the two bitmaps and inodes.
- **The Inode Table:**
- Physical Disk capacity in our example (64 blocks of 4KB each): 4 x 64 = 256 KB
- Logical capacity (8 blocks are reserved): 4 x 56 = 224 KB (the actual data storage space)
- Maximum number of inodes (each inode is 256 bytes): (5 * 4 * 1024) / 256 = 80 inodes (i.e max number of files)
- Size of the inode bitmap (1 bit per inode): 1 x 80 inodes = 80 bits (out of 32K bits)
- Size of the data bitmap (1 bit per storage block): 1 bit x 56 blocks = 56 bits (out of 32K bits, max data storage 128 MB)
- Decoding inodes: E.g. What disk sector to read to retrieve inode 32?
    1. Calculate the offset (each inode is 256 bytes) 32 x 256 = 8,192
    2. Add the start of the address of the inode table (12K) 8,192 + 12 x 1,024 = 20,480 (20 KB)
    3. Find the corresponding disk sector (each sector is 512 bytes) (20 x 1,024) / 512 = 40

- Simplified Unix Inode Table:



Size	Name	Description
2	mode	can the file be read/written/executed
2	uid	file owner id
4	size	the file size in bytes
4	time	time the file was last accessed
4	ctime	time when the file created
4	mtime	time when the file was last modified
4	dtime	time when the inode was deleted
2	gid	file group owner id
2	links_count	number of hard links pointing to this file
4	blocks	the number of blocks allocated to this file
60	block	disk pointers (15 in total)
4	file_acl	ACL permissions
4	dir_acl	ACL permissions

- So far, each inode has 15 block pointers. This means that the maximum file size can be 60KB (15 * 4 KB = 60 KB). Remember, 1 block is 4KB. Should we increase the number of block pointers to increase the file size?
    - Large file size with lots of unused entries means the mapping table requires a large chunk of contiguous space. A solution is to use a mapping table structured as a multi-level index array. This is called **multi-level indexed files**.
    - Unix uses 12 pointers that are direct blocks, which solves the problem of the first blocks' access being slow. Then it uses single, double, and triple indirect block pointers.



- File size with multi-level indexed files:
- File size using 12 direct blocks: 12 x 4 KB = 48 KB
    - Adding single indirect block: (12 + 1024) x 4 KB ~ 4 MB
    - Adding a double indirect block: (12 + 1024 + 1024^2) × 4 KB) ~ 4 GB
    - Adding a triple indirect block: (12 + 1024 + 1024^2 + 1024^3) × 4 KB) ~ 4 TB
- Rationale behind multi-level index files:
    - Most files are small: ~2K is the most common size
    - Average file size is growing: Almost 200K is the average
    - Most bytes are stored in large files: A few big files use most of space
    - File systems contains lots of files: Almost 100K on average
    - File systems are roughly half full: Even as disks grow, file systems remain ~50% full
    - Directories are typically small: Many have few entries; most have 20 or fewer

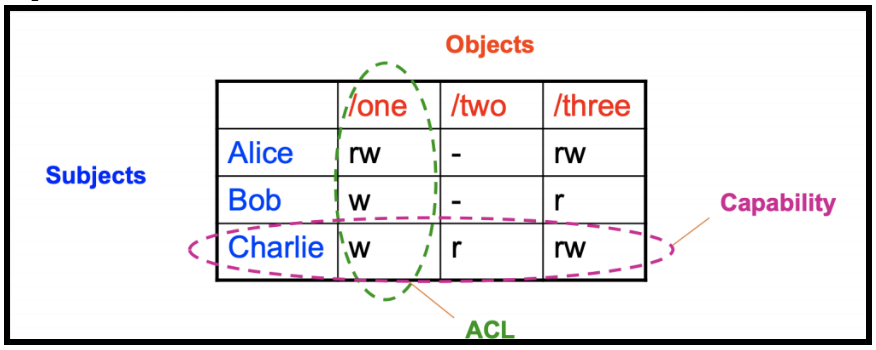
- **Directories:**
- Directories serve two purposes:
    1. For users, they provide a structured way to organize files by using digestible names rather than inode numbers directly.
    2. For the File System, they provide a convenient naming interface that allows the separation of logical file organization from physical file placement on the disk.
- Basic Directory Operations:

Unix	**Windows**
Directories implemented in file and a C runtime library provides a higher-level abstraction for reading directories.	Explicit directory operations.
opendir(name)	CreateDirectory(name)
readdir(DIR)	RemoveDirectory(name)
seekdir(DIR)	FindFirstFile(pattern)
closedir(DIR)	FindNextFile()

- A Short History of Directories:
    1. Approach 1: Single directory for the entire system.
        - Puts a directory at a known location on the disk.
        - Directories contain hname, inumberi pairs.
        - If one user uses a name, no one else can.
        - Many ancient personal computers work this way.
    2. Approach 2: Single directory for each user.
        - Still clumsy, and ls on 10,000 files is a real pain.
    3. Approach 3: Hierarchical name spaces.
        - Allow the directory to map names to files or other directories.
        - File system forms a tree (or graph, if links are allowed).
        - Large name spaces tend to be hierarchical (ip addresses, domain names, scoping in programming languages, etc.)
- Hierarchical Directory:
    - Used since CTSS (1960s) Unix picked up and used really nicely.
    - Directories stored on disk just like regular files:
        - A special inode type byte set to directory.
        - Users can read just like any other file.
        - Only special syscalls can write.
        - Inodes are at a fixed disk location.
        - A file pointed to by the index may be another directory.
        - Makes the file system into a hierarchical tree.
    - Simple, plus speeding up file ops speeds up dir ops.
- **Note:** Unix inodes are not directories. Inodes describe where on the disk the blocks for a file are placed whereas directories are files. So, inodes also describe where the blocks for directories are placed on the disk.

- Directory entries map file names to inodes:
    1. To open "/one", use Master Block to find the inode for "/" on the disk.
    2. Open "/", look for the entry for "one”.
    3. This entry gives the disk block number for the inode for "one".
    4. Read the inode for "one" into memory.
    5. The inode says where the first data block is on disk.
    6. Read that block into memory to access the data in the file.
- In Unix, each process has a "current working directory" (cwd). File names not beginning with "/" are assumed to be relative to cwd. Otherwise the translation happens as before.
- Shells track a default list of active contexts.
    - Given a search path A:B:C, the shell will check in A, then B, then C.
    - We can escape using explicit paths. For example: "./foo".
- Hard and Soft Links (synonyms):
    - More than one directory entry can refer to a given file.
    - A **hard link** creates a synonym for file.
    - Unix stores count of pointers ("hard links") to inode.
    - If one of the links is removed, the data is still accessible through any other link that remains.
    - If all links are removed, the space occupied by the data is freed.
    - **Soft symbolic links** are synonyms for names.
    - Soft links point to a file/dir name, but objects can be deleted from underneath it (or never exist).
    - Unix implements soft links like directories. The inode has a special "symlink" bit set and contains the name of the link target.
    - When the file system encounters a soft link it automatically translates it if possible.
- **File Buffer Cache:**
- Applications exhibit significant locality for reading and writing files.
- Idea: Cache file blocks in memory to capture locality called the **file buffer cache**.
    - Cache is system wide, used and shared by all processes.
    - Reading from the cache makes a disk perform like memory.
    - Even a small cache can be very effective.
- Issues:
    - The file buffer cache competes with VM (tradeoff here).
    - Like VM, it has limited size.
    - Need replacement algorithms again (LRU is usually used).
- Caching Writes:
    - On a write, some applications assume that data makes it through the buffer cache and onto the disk. As a result, writes are often slow even with caching.
    - OSes typically do write back caching:
        - Maintain a queue of uncommitted blocks.
        - Periodically flush the queue to disk (30 second threshold).
        - If blocks have changed many times in 30 secs, we only need one I/O.
        - If blocks are deleted before 30 secs (e.g., /tmp), no I/Os needed.
    - This is unreliable, but practical:
        - On a crash, all writes within the last 30 secs are lost.
        - Modern OSes do this by default; too slow otherwise.
        - System calls (Unix: fsync) enable apps to force data to disk.

- **Read Ahead:**
- Many file systems implement "read ahead".
    - The FS predicts that the process will request the next block.
    - The FS goes ahead and requests it from the disk.
    - This can happen while the process is computing on the previous block.
    - Overlap I/O with execution.
    - When the process requests a block, it will be in cache.
    - Compliments the disk cache, which also is doing read ahead.
- For sequentially accessed files read ahead can be a big win. Read ahead won't work if the blocks for the file are scattered across the disk, but file systems try to prevent that, though during allocation.
- **File Sharing:**
- File sharing is important for getting work done. It is the basis for communication and synchronization.
- There are two key issues when sharing files:
    1. Semantics of concurrent access:
        - What happens when one process reads while another writes?
        - What happens when two processes open a file for writing?
        - What are we going to use to coordinate?
    2. Protection
- **Protection:**
- File systems need to implement a protection system.
    - Who can access a file?
    - How can they access it?
- A protection system dictates whether a given action performed by a given subject on a given object should be allowed.
    - I.e. You can read and/or write your files, but others cannot.
    - I.e. You can read "/etc/motd", but you cannot write it.
- **Access Control Lists (ACL):** For each object, maintain a list of subjects and their permitted actions.
- **Capabilities:** For each subject, maintain a list of objects and their permitted actions.
- E.g.



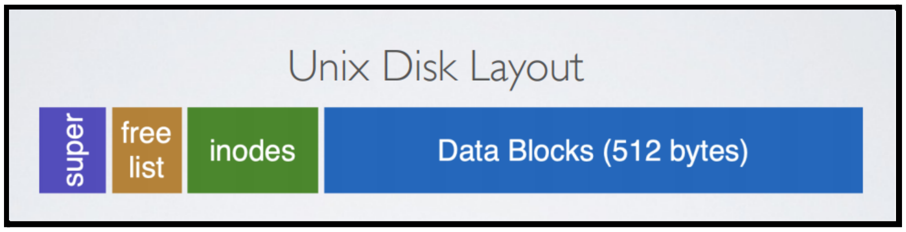
	/one	/two	/three
Alice	rw	-	rw
Bob	w	-	r
Charlie	w	r	rw

- An ACL is a list for each object consisting of the domains with a nonempty set of access rights for that object. A capability list is a list of objects and the operations allowed on those objects for each domain.
- E.g. Consider this scenario: There is a list of buildings and a list of people and we want to have a list of which people can enter which buildings. There are 2 ways to do it:
    1. ACL: Each building has a list of people that can enter it.
    2. Capability: Each person has a list of buildings that they can enter.

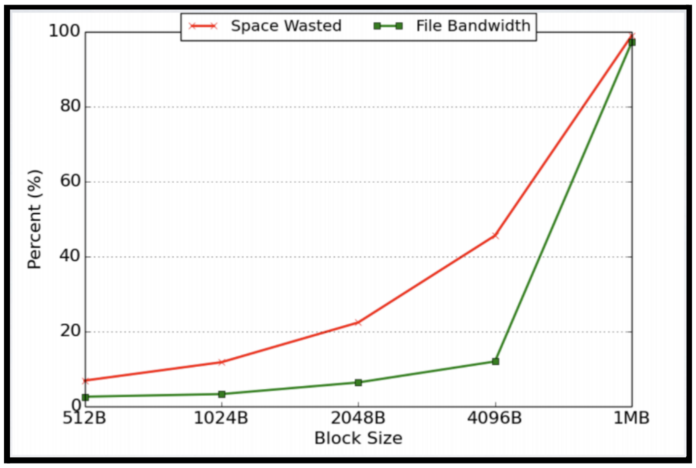
- Approaches between ACL and capability differ only in how the table is represented.
- Capabilities are easier to transfer. They are like keys, can handoff, and do not depend on the subject.
- ACLs are easier to manage in practice. They are object-centric, easy to grant, and revoke. To revoke capabilities, we have to keep track of all subjects that have the capability, which is a challenging problem.
- However, ACLs have a problem when objects are heavily shared, they become very large.

Lecture Notes:

- **Improving Performances with BSD Fast File System:**
- This is the original Unix FS layout:



- It is slow on hard disk drive - only gets 2% of disk maximum (20Kb/sec) even for sequential disk transfers.
- There were 3 problems to why it was so slow:
    1. In the original Unix File System, the blocks were too small (512 bytes).
    - Because the file index was too large, it required more indirect blocks but the transfer rate was low (get one block at time).
    2. Unorganized freelist:
    - Consecutive file blocks are not close together. This meant the OS had to pay a seek cost for even sequential access. Another issue was **aging**, which is when the freelist becomes fragmented over time.
    3. Poor locality:
    - The inodes were far from data blocks.
    - Furthermore, inodes for directories were not close together. This meant poor enumeration performance, for commands like ls.
- **Problem 1 - blocks are too small:**
- Bigger block increases bandwidth, but increases internal fragmentation as well.
- E.g.



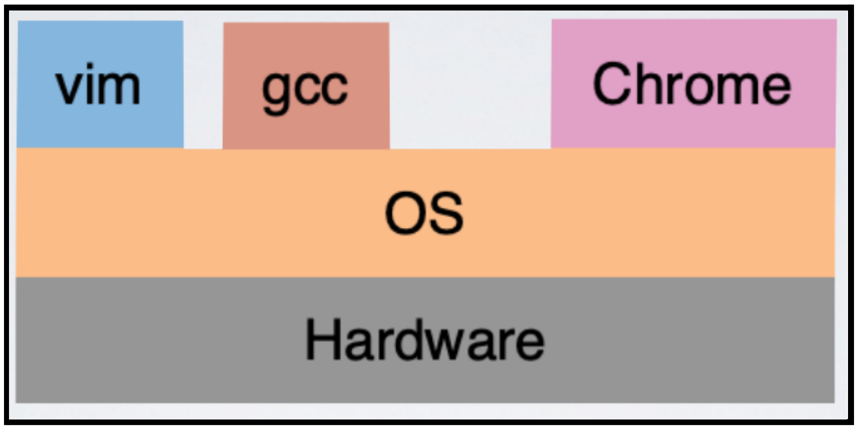
- The solution is to use fragments.
- Allow large blocks to be chopped into small ones called "fragments".
- Ensure fragments are only used for little files or ends of files.
- Ensure that the fragment size is specified at the time that the file system is created.
- Limit the number of fragments per block to 2, 4, or 8.
- This solution has a high transfer speed for larger files and low wasted space for small files or ends of files.

- **Problem 2 - Unorganized Freelist:**
- Unorganized freelist leads to random allocation of sequential file blocks overtime.
- The solution is to use bitmaps.
- Periodical compact/defragment disk but locks up disk bandwidth during operation.
- Keep adjacent free blocks together on the freelist but costly to maintain,
- Each bit indicates whether the block is free.
- Easier to find contiguous blocks.
- Are small, so usually keep the entire thing in memory.
- The time to find free blocks increases if there are fewer free blocks.
- Here's the algorithm:
    - Allocate a block close to block x.
    - Check for blocks near bmap[x/32].
    - If the disk is almost empty, it will be likely to find one near.
    - As the disk becomes full, search becomes more expensive and less effective.
- **Problem 3 - Poor Locality:**
- The solution is to use a **cylinder group**.
- Group sets of consecutive cylinders into cylinder groups.
- Using this way, the OS can access any block in a cylinder without performing a seek as the next fastest place is the adjacent cylinder.
- Tries to put everything related in the same cylinder group.
- Tries to put everything not related in different groups.
- If you access one block, you'll probably access the next block, too, so let's try to put sequential blocks in adjacent sectors.
- If you look at an inode, you'll most likely look at the file data too, so let's try to keep the inode in the same cylinder as the file data.
- If you access one file in a directory, you'll probably frequently access the other files in that directory, so let's try to keep all inodes in a directory in the same cylinder group.
- How to keep inode close to the data block? → Use groups across disks and allocate inodes and data blocks in the same group. This way, each cylinder group is basically a mini-Unix file system.
- **Improving Reliability with Log-Structured File system (LFS) and Journaling File System (ext3):**
- What happens when there's power loss or a system crash:
    - Sectors (but not a block) are written atomically by the hard drive device.
    - But an FS operation might modify several sectors, such as metadata blocks (free bitmaps and inodes) and data blocks. Hence, a crash has a high chance of corrupting the file system.
- Solution 1 - Unix fsck (File System Checker):
    - When the system boots, check the system looking for inconsistencies and try to fix errors automatically.
    - However, this cannot fix all crash scenarios.
    - Furthermore, it has poor performance. Sometimes it takes hours to run on large disk volumes and does fsck have to run upon every reboot (Not well-defined consistency)?
- Solution 2 - Log Structure File System (LFS) or Copy-On-Write Logging:
    - The idea is to treat the disk like a tape-drive.
    - Buffer all data (including inode) in memory segment.
    - Write buffered data to a new segment on disk in a sequential log.
    - Existing data is not overwritten as the segment is always written in a free location.
    - Best performance from disk for sequential access.

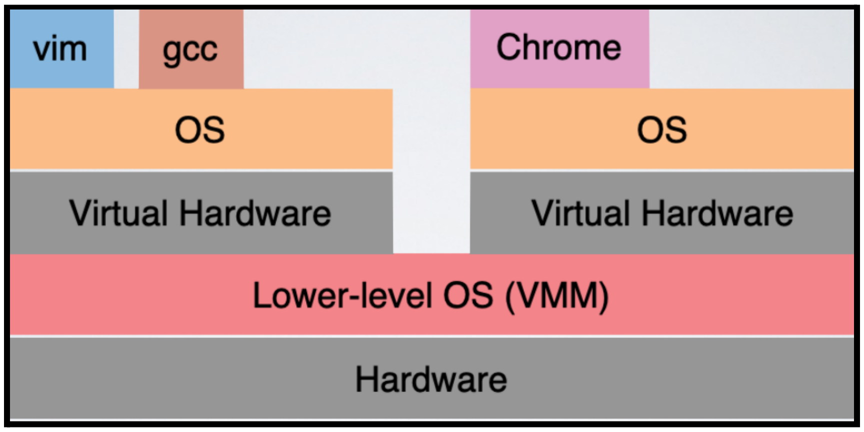
    - In the original Unix File System, the inode table is placed at a fixed location. However, in a Log-structured File System, the inode table is split and spread-out on the disk. Hence, the LFS needs to use an inode map (imap) to map the inode number with its location on disk.
    - The OS must have some fixed and known location on disk to begin a file lookup. The check-point region (CR) contains a pointer to the latest pieces of the inode map. The CR is updated periodically (every 30 sec or so) to avoid degrading the performances.
    - LFS - Crash recovery:
        - The check-point region (CR) must be updated atomically.
        - The LFS keeps two CRs and writing a CR is done in 3 steps:
            1. Writes out the header with a timestamp #1.
            2. Writes the body of the CR.
            3. Writes one last block with another timestamp #2.
        - A crash can be detected if timestamp #1 is after #2.
        - The LFS will always choose the most recent and valid CR.
        - All logs written after a successful CR update will be lost in case of a crash.
    - LFS - Disk Cleaning (a.k.a Garbage Collection):
        - The LFS leaves an old version of file structures on disk.
        - The LFS keeps information of the version of each segment and runs a disk cleaning process.
        - A cleaning process removes old versions by compacting contiguous blocks in memory.
        - That cleaning process runs when the disk is idle or when running out of disk space.
- Solution 3 - Journaling or Write-Ahead Logging:
    - Write the "intent" down to disk before updating the file system.
    - When a crash occurs, look through the log to see what was going on and use the contents of the log to fix file system structures.
    - The process is called **recovery**.

__Lecture Notes:__

- **Virtualization:**
- What is an OS:
    - An OS is software between applications and hardware.
    - It abstracts the hardware to make applications portable.
    - It makes finite resources (such as memory, number of CPU cores) appear much larger and fully dedicated to running one application.
    - It protects processes and users from one another.
    - I.e.



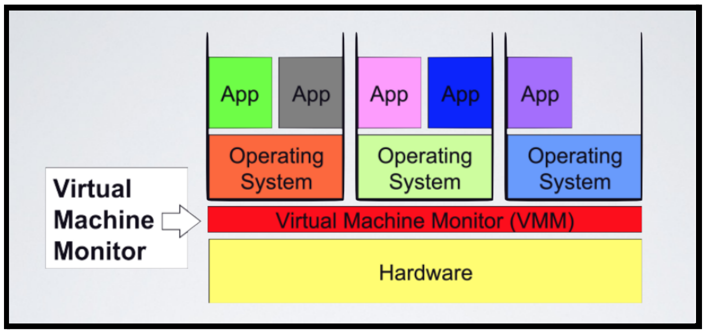
    - However, what if the process abstraction looked just like hardware? I.e.



- How do process abstraction & hardware differ:

Process	**Hardware**
Non-privileged registers and instructions	All registers and instructions
Virtual memory	Both virtual and physical memory, MMU functions, TLB/page tables, etc
Errors and signals	Trap, interrupts
File systems, directories, files, raw devices	I/O devices accessed through programmed I/O, DMA, interrupts

- VMM - Virtual Machine Monitor:
    - The VMM is a thin layer of software that virtualizes the hardware.
    - It exports a virtual machine abstraction that looks like the hardware.
    - It provides the illusion that software has full control over the hardware.
    - The **Virtual Machine Monitor/Hypervisor** runs multiple OSes simultaneously on the same physical machine.
    - I.e.

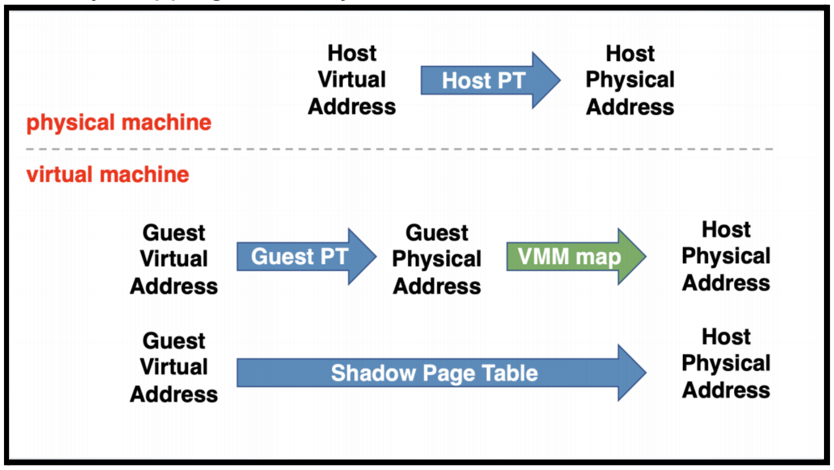


- **Motivations:**
- Virtualization was an old idea, starting from the 70's. At that time, computers were very big (they took up a room). However, in the 80's computers began getting cheaper, so the idea died out. The 80's were the era of personal computers. However, the idea was revived by the Disco work led by Mendel Rosenblum, who later led the foundation of VMware.
- Nowadays, VMs are used everywhere.
- They got popularized by cloud computing and are used to solve different problems.
- VMMs are a hot topic in industry and academia.
- Why we use virtualization:
    1. **Software compatibility:** VMMs can run pretty much all software.
    2. **Resource utilization:** Machines today are powerful, and we want to multiplex their hardware.
    3. **Isolation:** Seemingly total data isolation between virtual machines.
    4. **Encapsulation:** Virtual machines are not tied to physical machines.
    5. **Many other cool applications:** Debugging, emulation, security, speculation, fault tolerance, etc.
- Backward compatibility is the bane of new OSes as it requires huge efforts to innovate but not break. Security considerations may make it impossible in practice.
- Logical partitioning of servers:
    - **Run multiple servers on same box (e.g. Amazon EC2):**
        - Modern CPUs are much more powerful than most services need.
        - VMs let you give away less than one machine for running a service.
        - Server consolidation: N machines → 1 real machine.
        - Consolidation leads to cost savings (less power, cooling, management, etc).
    - **Isolation of environments:**
        - Safety - A printer server failure doesn't take down Exchange server.
        - Security - The compromise of one VM cannot get the data of others.
    - **Resource management:**
        - Provide service-level agreements.
    - **Heterogeneous environments:**
        - Linux, FreeBSD, Windows, etc.

- **Implementation:**
- Implementing VMMs - requirements:
    - **Fidelity:** OSes and applications work the same without modification (although we may modify the OS a bit).
    - **Isolation:** VMM protects resources and VMs from each other.
    - **Performance:** VMM is another layer of software and therefore there is some overhead. We want to minimize the overhead.
- What needs to be virtualized:
    - Exactly what you would expect:
        - CPU
        - Events (hardware and software interrupts)
        - Memory
        - I/O devices
    - Isn't this just duplicating OS functionality in a VMM?:
        - Yes: Approaches will be similar to what we do with OSes. However, they will be simpler in functionality, since VMMs are much smaller than OSes.
        - No: It implements a different abstraction. Hardware interface vs. OS interface
- Approach 1 - Complete machine simulation:
    - Simplest VMM approach.
    - Used by Bochs.
    - The idea is to build a simulation of all the hardware:
        - CPU – A loop that fetches each instruction, decodes it, and simulates its effect on the machine state (no direct execution).
        - Memory – Physical memory is just an array. Simulate the MMU on all memory accesses.
        - I/O – Simulate I/O devices, programmed I/O, DMA, interrupts, etc.
    - However, this is too slow.
      CPU/Memory – 100x slowdown.
      I/O Device – 2x slowdown.
    - We need faster ways of emulating the CPU/MMU.
- Approach 2 - Virtualizing the CPU/MMU:
    - Observations - Most instructions are the same regardless of processor privileged level.
      E.g. incl %eax.
    - Why not just give instructions to the CPU to execute?
    - The problem is safety. How can we prevent privileged instructions from interfering with the hypervisor and other OSes?
    - The solution is to use the protection mechanisms already in the CPU.
    - I.e. "Trap and emulate" approach:
        - Run the virtual machine's OS directly on the CPU in unprivileged user mode.
        - Privileged instructions trap into monitor and run simulator on instruction.
- Virtualizing interrupts:
    - The OS assumes to be in control of interrupts via the interrupt table.
    - So what happens when an interrupt or trap occurs in a virtual environment? ➡ The VMM handles the interrupt (in kernel mode) using the "virtual" interrupt handler table of the running OS.
    - Some interrupts can be shadowed.

- Virtualizing memory:
    - The OS assumes to be in full control over memory via the page table.
    - But VMM partitions memory among VMs:
        - VMM needs to assign hardware pages to VMs.
        - VMM needs to control mappings for isolation.
          Cannot allow an OS to map a virtual page to any hardware page.
          The OS can only map to a hardware page given to it by the VMM.
    - Hardware-managed TLBs make this difficult. When the TLB misses, the hardware automatically walks the page tables in memory. As a result, the VMM needs to control access by OS to page tables.
    - One way - direct mapping:
        - The VMM uses the page tables that a guest OS creates (direct mapping by MMU).
        - The VMM validates all updates to page tables by guest OS. The OS can read page tables without modification but the VMM needs to check all page table entry writes to ensure that the virtual-to-physical mapping is valid.
        - This requires the OS to patch updates to the page table.
    - Another way - level of indirection:
        - Three abstractions of memory:
            1. Machine - Actual hardware memory (16 GB of DRAM).
            2. Physical - Abstraction of hardware memory managed by OS. If a VMM allocates 512 MB to a VM, the OS thinks the computer has 512 MB of contiguous physical memory (underlying machine memory may be discontiguous).
            3. Virtual - Virtual address spaces (similar to virtual memory). The standard is 2^{32} or 2^{64} address space.
    - Shadow page tables:
        - The VMM creates and manages page tables that map virtual pages directly to machine pages. These tables are loaded into the MMU on a context switch.
        - The VMM page tables are the shadow page tables.
        - The VMM needs to keep its virtual to machine tables consistent with changes made by the OS to its virtual to physical tables.
            - VMM maps OS page tables as read-only (i.e., write-protected).
            - When the OS writes to page tables, trap to VMM.
            - Memory tracing: VMM applies write to shadow table and OS table and returns.
            - Memory-mapped devices must be protected for both read-protected and write-protected.

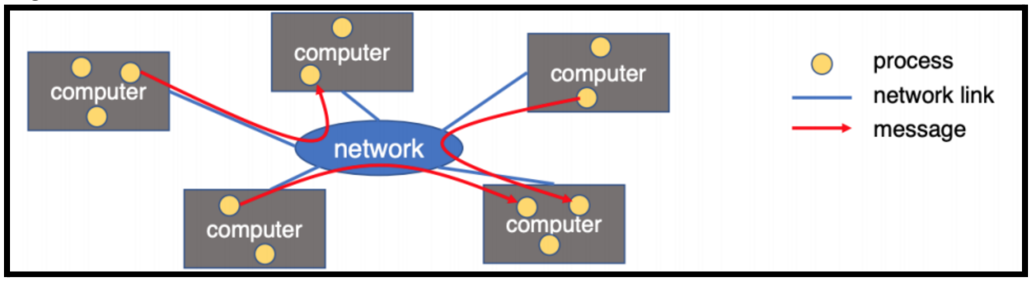
- Memory mapping summary:



- Memory Allocation:
  - VMMs tend to have simple hardware memory allocation policies:
    - Static - VM gets 512 MB of hardware memory for life.
    - No dynamic adjustment based on load. OSes are not designed to handle changes in physical memory.
    - No swapping to disk.
  - More sophistication - overcommit with balloon driver:
    - Balloon driver runs inside the OS to consume hardware pages and steals from virtual memory and file buffer cache (balloon grows).
    - Gives hardware pages to other VMs (those balloons shrink).
- Virtualizing I/O:
  - OSes can no longer interact directly with I/O devices.
    1. Make an in/out trap into VMM and use tracing for memory-mapped I/O.
    2. Run simulation of I/O device.
       Interrupt – tell CPU simulator to generate interrupt.
       DMA – copy data to/from physical memory of virtual machine.

Lecture Notes:

- **Distributed Systems:**
- A **distributed system** is cooperating processes in a computer network.
- It is a group of computers working together as to appear as a single computer to the end-user. These machines have a shared state, operate concurrently and can fail independently without affecting the whole system's uptime.
- E.g.



- Some popular distributed systems today include:
    - Google file systems
    - BigTable
    - MapReduce
    - Hadoop
    - ZooKeeper
- There are 3 degrees of integration for distributed systems:
    1. **Loosely-coupled:** E.g. internet applications (email, web, FTP, SSH).
    2. **Mediumly-coupled:** E.g. remote execution (RPC), remote file system (NFS).
    3. **Tightly-coupled distributed:** E.g. file systems (AFS)
- Advantages of distributed systems:
    1. Performance - parallelism across multiple nodes.
    2. Scalability - by adding more nodes.
    3. Reliability - leverage redundancy to provide fault tolerance.
    4. Cost - cheaper and easier to build lots of simple computers.
    5. Control - users can have complete control over some components.
    6. Collaboration - much easier for users to collaborate through network resources.
- The promise of distributed systems:
    1. Higher availability - when one machine goes down, use another.
    2. Better durability - store data in multiple locations.
    3. More security - each piece is easy to secure.
- The reality of distributed systems:
    1. Worse availability - depend on every machine being up.
    2. Worse reliability - can lose data if any machine crashes.
    3. Worse security - anyone in the world can break into the system.

  Coordination is more difficult - must coordinate multiple copies of shared state information (using only a network).

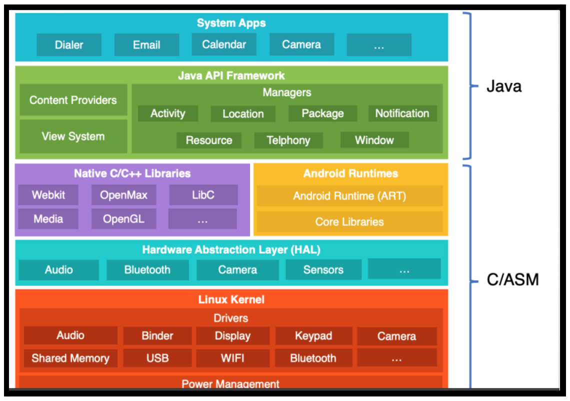
- Requirements:
  - **Transparency:** The ability of the system to mask its complexity behind a simple interface.
  - Possible transparencies:
    - Location - cannot tell where resources are located.
    - Migration - resources may move without the user knowing.
    - Replication - cannot tell how many copies of resources exist.
    - Concurrency - cannot tell how many users there are.
    - Parallelism - may speed up large jobs by splitting them into smaller pieces.
    - Fault Tolerance - system may hide various things that go wrong.
  - Transparency and collaboration require some way for different processors to communicate with one another.
- Clients and Servers:
  - The prevalent model for structuring distributed computation is the client/server paradigm.
  - A **server** is a program or collection of programs)that provides a service.
  - The server may exist on one or more nodes.
  - **Note:** Often the node is called the server, too, which is confusing.
  - A **client** is a program that uses the service.
  - A client first binds to the server (locates it and establishes a connection to it). Then, the client sends requests, with data, to perform actions, and the server sends responses, also with data.
- Naming:
  - Essential naming systems in network:
  - Address processes/ports within the system (host, id) pair.
  - Physical network address (Ethernet address).
  - Network address (Internet IP address).
  - Domain Name Service (DNS) provides resolution of canonical names to network addresses.
- Communication:
  - There are a few ways computers can communicate with each other:
  1. **Raw Message - UDP:**
     - Network programming = raw messaging (socket I/O).
     - Programmers hand-coded messages to send requests and responses.
     - This method is too low-level and tiresome.
     - Need to worry about message formats.
     - Must wrap up information into a message at source.
     - Must decide what to do with the message at the destination.
     - Have to pack and unpack data from messages.
     - May need to sit and wait for multiple messages to arrive.
  2. **Reliable Message - TCP:**
  3. **Remote Procedure Call (RPC) and Remote Method Invocation(RMI):**
     - **Procedure calls** are a more natural way to communicate.
     - Every language supports them.
     - Semantics are well-defined and understood.
     - Natural for programmers to use.
     - The idea is to let servers export procedures that can be called by client programs.
     - Similar to module interfaces, class definitions, etc.

- Clients just do a procedure call as if they were directly linked with the server.
- Under the covers, the procedure call is converted into a message exchange with the server.
- **Remote Procedure Call (RPC)** is the most common means for remote communication.
- It is used both by operating systems and applications.
- DCOM, CORBA, Java RMI, etc., are all basically just RPC. NFS is implemented as a set of RPCs.
- A server defines the server's interface using an **Interface Definition Language (IDL)** that specifies the names, parameters, and types for all client-callable server procedures.
- A **stub compiler** reads the IDL and produces two stub procedures for each server procedure (client and server).
- Server programmer implements the server procedures and links them with server-side stubs.
- Client programmer implements the client program and links it with client-side stubs.
- The **stubs** are the "glues" responsible for managing all details of the remote communication between client and server. They send messages to each other to make RPC happen transparently.
  A client-side stub packs the message, sends it off, waits for the result, unpacks the result and returns to the caller.
  A server-side stub unpacks the message, calls the procedure, packs the results, sends them off.
- **Marshalling** is the packing of procedure parameters into a message packet.
- The RPC stubs call type-specific procedures to marshal or unmarshal the parameters to a call.
  The client stub marshals the parameters into a message.
  The server stub unmarshals parameters from the message and uses them to call the server procedure.
- On return:
    - The server stub marshals the return parameters.
    - The client stub unmarshals return parameters and returns them to the client program.

<u>Lecture Notes:</u>

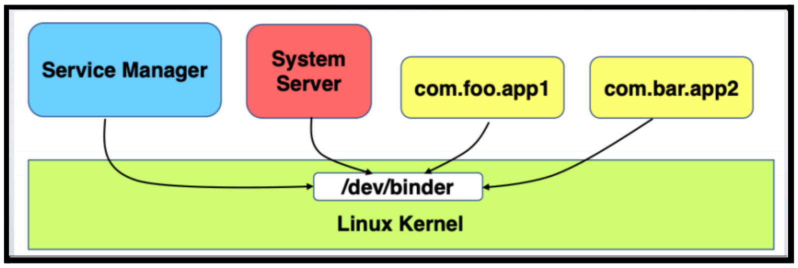
- **Mobile OS:**
- History of mobile OSes:
    - Early smart devices are PDAs (touchscreen, Internet).
    - Symbian is the first modern mobile OS. It was released in 2000 and ran in Ericsson R380, the first “smartphone" (mobile phone + PDA). It only supported proprietary programs.
    - Many smartphone and mobile OSes followed after:
        - Palm OS (2001)
        - Windows CE (2002)
        - Blackberry (2002)
        - Introduction of iPhone (2007) ← This was a game changer
          It had 4GB flash memory, 128 MB DRAM, and multi-touch interface. Originally, it only ran iOS proprietary apps but the App Store opened in 2008 and allowed third party apps.
- Design considerations for mobile OS:
    - Resources are very constrained:
        - Limited memory
        - Limited storage
        - Limited battery life
        - Limited processing power
        - Limited network bandwidth
        - Limited size
    - User perception are important: Latency ≫ throughput.
      Users will be frustrated if an app takes several seconds to launch.
    - The environment is frequently changing. Cellular signals change from strong to weak and then back to strong.
- Process management in mobile OS:
    - On a desktop/server, an application = a process. This is not true on mobile OSes.
    - On mobile OSes:
        - When you see an app present to you it does not mean an actual process is running.
        - Multiple apps might share processes.
        - An app might make use of multiple processes.
        - When you close an app, the process might be still running.
    - Multitasking is a luxury in mobile OS.
    - Early versions of iOS did not allow multi-tasking mainly because of battery life and limited memory.
    - Only one app runs in the foreground. All the other user apps are suspended.
    - The OS's tasks are multi-tasked because they are assumed to be well-behaving. Starting with iOS 4, the OS APIs allow multitasking in apps but are only available for a limited number of app types.
- Memory management in mobile OS:
    - Most desktop and server OSes today support swap space.
    - Mobile OSes typically do not support swapping.
    - iOS asks applications to voluntarily relinquish allocated memory.
    - Android will terminate an app when free memory is running low.
    - App developers must be very careful about memory usage.

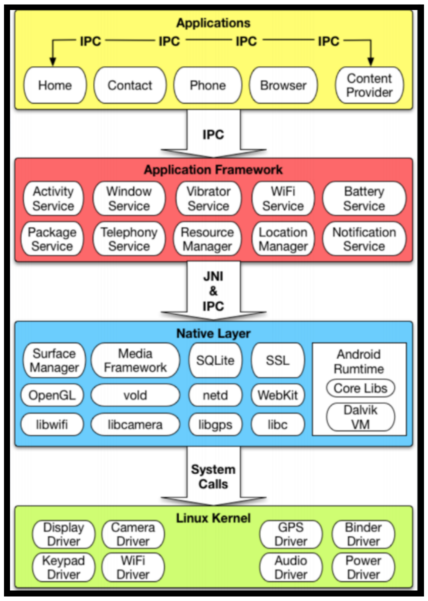
- Storage in mobile OS:
    - App privacy and security is hugely important in mobile devices.
    - Each app has its own private directory that other apps cannot access.
    - The only shared storage is external storage.
- Android:
    - History of Android:
        - Android Inc was founded by Andy Rubin et al. in 2003.
        - The original goal is to develop an OS for a digital camera.
        - Later, the focus shifted on Android as a mobile OS.
        - Android was later bought by Google.
        - Originally, no carrier wanted to support it except for T-Mobile. While preparing for the public launch of Android, the iPhone was released.
        - Android 1.0 was released in 2008 (HTC G1).
        - In 2019, Android has ~87% of the mobile OS market while iOS has ~13%.
    - Android OS stack:



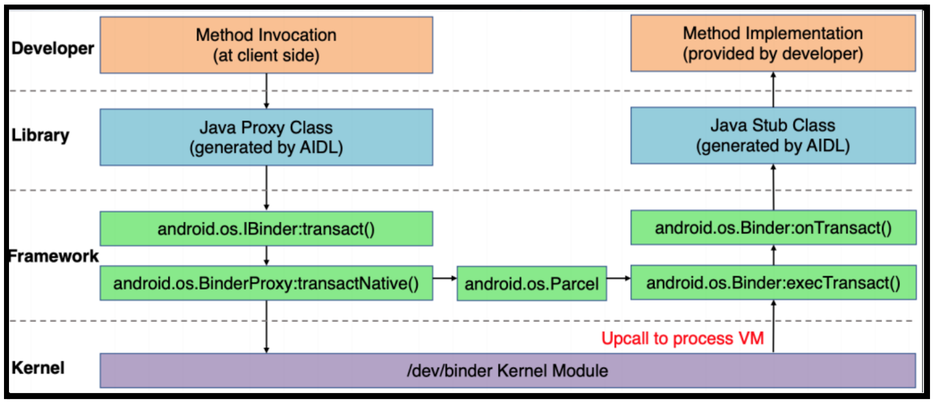
-
    - Linux kernel vs. Android kernel:
        - The Linux kernel is the foundation of the Android platform.
        - However, there are a few tweaks:
            - binder - interprocess communication mechanism
            - shmem - shared memory mechanism
            - logger
    - Android runtime:
        - **Runtime:** A component provides functionality necessary for the execution of a program.
          E.g. scheduling, resource management, stack behavior, etc
        - Prior to Android 5.0 (Dalvik):
            - Each Android app has its own process, runs its own instance of the Dalvik virtual machine (process virtual machine).
            - The VM executes the Dalvik executable (.dex).
            - The Dalvik virtual machine is register-based compared to stack-based of a JVM.
        - After Android 5.0 (ART):
            - Backward compatible for running Dex bytecode.
            - New feature - Ahead-Of-Time (AOT) compilation.
            - Improved garbage collection.

- Android process creation:
    - All Android apps derive from a process called Zygote.
    - Zygote is started as part of the init process.
    - It preloads Java classes, resources, and starts the Dalvik VM.
    - It registers a Unix domain socket.
    - It waits for commands on the socket.
    - It forks off child processes that inherit the initial state of VMs. It uses Copy-on-Write only when a process writes to a page will a page be allocated.
- Java API framework:
    - The main Android OS from app point of view.
    - It provides high-level services and environment to apps.
    - It interacts with low-level libraries and Linux kernels.
    - Some components:
        - Activity Manager - manages the lifecycle of apps.
        - Package Manager - keeps track of apps installed.
        - Power Manager - wakelock APIs to apps.
- Native C/C++ libraries:
    - Many core Android services are built from native code.
    - They require native libraries written in C/C++.
    - Some of them are exposed through the Java API framework as native APIs such as the Java OpenGL API
- Android Binder IPC:
    - Android Binder IPC allows communication among apps, between system services, and between app and system service.





- Binder is implemented as an RPC:
    1. Developer defines methods and object interface in an .aidl file.
    2. Android SDK generates a stub Java file for the .aidl file and exposes the stub in a Service.
    3. Developer implements the stub methods.
    4. Client copies the .aidl file to its source.
    5. Android SDK generates a stub (a.k.a proxy).
    6. Client invokes the RPC through the stub.
- Binder information flow:


Developer
Method Invocation (at client side)
Method Implementation (provided by developer)
Library
Java Proxy Class (generated by AIDL)
Java Stub Class (generated by AIDL)
Framework
android.os.IBinder:transact()
android.os.Binder:onTransact()
android.os.BinderProxy:transactNative()
android.os.Parcel
android.os.Binder:execTransact()
Upcall to process VM
Kernel
/dev/binder Kernel Module

- **OS Security:**
- Protection:
    - File systems implement a protection system:
        - Who can access a file?
        - How can they access it?
    - A protection system dictates whether a given action performed by a given subject on a given object should be allowed.
      E.g. You can read and/or write your files, but others cannot.
      E.g. You can read "/etc/motd", but you cannot write it.
- DAC vs MAC:
    - **DAC (Discretionary Access Control):** Users define their own policy on their own data.
    - **MAC (Mandatory Access Control):** The administrator defines a system level policy to control the propagation of data between users.
    - DAC and MAC are not exclusive and can be used together.
- Discretionary Access Control:
    - Unix protection on files:
        - Each process has a User ID and one or more group IDs.
        - The system stores the following with each file:
            - The user who owns the file and the group the file is in.
            - Permissions for users, any one in the file group, and other.
        - This is shown by the output of the "ls -l" command.
    - Unix protection on directories:
        - Directories have permission bits, too.
        - The write permission on a directory allows users to create or delete a file.
        - The execute permission allows users to use pathnames in the directory.
        - The read permission allows users to list the contents of the directory.
        - The special user root (UID 0) has all privileges. It is required for administration.

- Unix permissions on non-files:
    - Many devices show up in the file system.
      E.g. /dev/tty1
    - They have permissions just like for files. However, other access controls are not represented in the file system.
      E.g. You must usually be root to do the following:
        - Bind any TCP or UDP port number less than 1024.
        - Change the current process's user or group ID.
        - Mount or unmount most file systems.
        - Create device nodes (such as /dev/tty1) in the file system.
        - Change the owner of a file.
        - Set the time-of-day clock; halt or reboot machine.
- Setuid:
    - Some legitimate actions require more privileges than UID.
      E.g. how users change their passwords stored in root-owned /etc/passwd and /etc/shadow files?
    - The solution is the setuid and setgid programs.
      Run with privileges of the file's owner or group.
      Each process has a real and effective UID/GID.
      Real is a user who launched the setuid program.
      Effective is the owner/group of the file, used in access checks.
    - Have to be very careful when writing setuid code.
      Attackers can run setuid programs any time (no need to wait for root to run a vulnerable job).
      Attacker controls many aspects of the program's environment.
- Unix security hole: Even without root or setuid attackers can trick root owned processes into doing things.

- Mandatory Access Control:
    - Mandatory access control (MAC) can restrict propagation.
      E.g. A security administrator may allow you to read but not disclose the file.
    - MAC prevents users from disclosing sensitive information whether accidentally or maliciously.
      E.g. Classified information requires such protection.
    - MAC prevents software from surreptitiously leaking data. Seemingly innocuous software may steal secrets in the background (Trojan Horse).

GDB Notes

Introduction to GDB:

- GDB stands for GNU Debugger and is a debugger for several languages, including C and C++.
- To use GDB, you need to add the -g flag when you run gcc to compile your code.
- A **breakpoint** is like a stop sign in your code. Whenever gdb gets to a breakpoint it halts execution of your program and allows you to examine it.

Commands	**Description**
Help	
help	List the gdb command topics.
help *class*	List the gdb commands within the specified class.
help *command*	Give a description of the specified command.
Running	
run/r run/r *command-line arguments* run/r file *filename*	Start program execution from the beginning of the program.
q/quit	Exit GDB.
kill	Stop the program execution.
BreakPoints	
break *line-number/function-name*	Suspend the program at specified function of line number.
delete *breakpoint*	Deletes the specified breakpoint.
delete/d	Delete all breakpoints.
clear *function/line*	Delete all breakpoints in the given function or line.
Stepping	
continue/c	Continue executing until the next breakpoint.
next/n	Execute the next line of code. Will not enter functions.
step/s	Step to the next line of code. Will enter functions.
until *line-number*	Continue processing until you reach a specified line number. until is like next, except that if you are at the end of a loop, until will continue execution until the loop is exited, whereas next will just take you back up to the beginning of the loop. This is convenient if you want to see what happens after the loop, but don't want to step through every iteration.
finish	Continue until the current function returns.
where	Shows current line number and which function you are in.
print *variable-name*	Prints the value stored in the specified variable.
Stack	
up	Move up a single frame (element in the call stack)
down	Move down a single frame
up/down *number*	Move up/down by the specified number of frames in the stack.