Final Exam Study Guide

Wednesday, April 26, 2017 1:17 AM

PROCESSES

3.1 The Process Concept

- A process is a program that is currently in execution on a CPU/OS. This contains the program code, aka the text section, then the program counter/data in cpu registers, a data section with global variables, a stack which contains function parameters, return addresses, and local variables, and a heap, where dynamically allocated memory exists.
- A program can run simultaneously, but it will be considered two processes.
- A process can have many states. New means it is being created, running means the instructions are being executed, waiting means the process is waiting for I/O or other event, ready means a process is waiting to be assigned to a processor, and terminated means the process has finished.
- Every process has a process control block (PCB) that has many bits of info on a process, like its state, program counter value, last values of CPU registers, cpu scheduling info, and memory management info. It also has accounting info, like process numbers, amount of CPU used, among other things. It finally has I/O device info, like lists of open files and the I/O drivers assigned to the process.
- A thread is a logical object that contains a single running process at any time. Multicore systems have multiple thread support, which can simultaneously run processes.

3.2 Process Scheduling

- We want a process running at all times to ensure efficient use of the CPU. A process scheduler built into the OS will select processes to run at given points. When a process is put into a ready state, it enters a job queue, containing all processes in the OS. All processes are held in main memory. Usually implemented as a linked list, with a header pointing to the first and last ready PCB.
- Other queues exist, like I/O queue when waiting for I/O. If a process creates a child, the child should run and then the parent will resume, or interrupted and torn out of the cpu.
- A scheduler is a task that selects the PCBs from the job queue. A long term scheduler/job scheduler selects processes from hard disk and loads them into memory. A short term/CPU scheduler selects things from the ready queue and runs them. CPU scheduler executes usually every 100ms. The long term executes over minutes. The degree of multiprogramming is controlled by the long term, and is defined as the average rate of processes going into the ready queue vs the average rate of processes leaving the system, and is stable if these are roughly equal.
- Long term will select a good mix of both processes depending on I/O and processes deepening on CPU.
- Context switching is when a process is interrupted and the cpu is given to another process. A context is the state of the cpu at the time the process is taken away, and in a context switch, the context is saved in its processes PCB, and when loaded again the state is grabbed from the PCB.

3.3 Process Creation

- Processes can create other processes called children, and those form a tree. Processes are identified using process identifiers (pid), a unique value that can be used to index processes.
- A parent can either execute concurrently with its children, or wait until all or some of its kids have terminated. The child process will either be a duplicate of the parent, or have a new program loaded into it. Finally, a child will either get resources directly from the OS after creation, or will have to use a subset of its parents.
- A fork call in a C program in linux will create a child process with pid 0, and then that process can do whatever it wants, like call exec, which will load a new binary program from disk into memory. The parent will wait until the child dies, or continue executing concurrently.
- When a process terminates, it does this by calling the exit system call, and returns a status code in the form of an integer to its parent process. Then everything the process used is deallocated to the OS again. You can also kill a process manually, or programmatically, but only through the parent process. Cascading termination is the fact that most OS kill process children when a process is killed. The exit status is stored in a PCB, so a child process must wait until its parent calls

wait and checks for that status. A zombie process that has terminated but its parent hasn't called wait. An orphaned process is one that's parent never calls wait. Linux will assign the top level process, init, as the new parent for these, and kill em when ready.

MULTITHREADING

4.1 Overview

- A thread is a basic unit of CPU utilization, it has:
  - A thread ID
  - A Program Counter
  - A set of assigned registers
  - A stack
- It shares with other threads belonging to the same process:
  - Its code section.
  - Its data section
  - Other OS resources like open files and signals.

- A heavyweight process has only a single thread, but if it has multiple threads, it can perform multiple tasks at once.
- Instead of making one process that handles requests and spawns new processes to handle them, it is smarter to have them create threads all in one process. There are many benefits to a multithreaded application, like:
  - Responsiveness, which allows a program to continue running even if part of it is blocked by the user or I/O. Vital for GUIs..
  - Resource sharing, as threads share the memory and resources of the process they belong to by default, as they are in the same address space.
  - Economy, as process creation is costly, and which threads, resources are already shared with the parent process and it's more efficient to create and context switch threads.
  - Scalability, as threads can be run in parallel by running them on different cores.

4.2 Multicore Programming

- Every core in a single processor appears as a separate processor to the OS. On a system with 1 core, threads are all run interleaved over time, but on a multicore, a separate thread can be assigned to each core.
- A system can do parallelism if it can perform more than one task simultaneously, a system has concurrency if it supports more than one task at a time if all the tasks can make progress. A single core cannot be parallel, but it can be concurrent.
- There are five problems for app developers to solve to make programs more efficient with multithreading:
  - Identifying tasks that can be split up into separate concurrent tasks that are ideally independent of each other.
  - Balancing the work those tasks do, whether a task isn't doing too much or wasting a core by doing too little.
  - Splitting data so it will run on multiple cores by different tasks.
  - Synchronizing code (using chapter 5 strategies like semaphores) to ensure that when one task depends on data from another task, we can schedule the threads so that they run correctly.
  - Testing and debugging parallel threads is more difficult than a monolithic program.
- There are two types of parallelism:
  - Data, which focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core. Allows faster computation of large data sets.
  - Task, which distributes tasks across multiple cores that perform unique operations, possibly on the same data.

4.3 Multithreading Models

- Support for threads exists both at the user level, for user threads, and at the kernel level, for kernel threads. Kernel threads are managed by the OS, user threads are managed by applications

and libraries. Many models exist for mapping between the two.

- The many to one model maps many user level threads to one kernel thread. All threads will be blocked, however, if one thread makes a blocking system call, as they can all only use one thread. This also cannot do parallelism on a multicore processor.
- The one to one model maps each user thread to a kernel thread, it provides both concurrency and parallelism. Windows and Linux do this one. The only drawback is overhead in creating kernel threads for every user thread.
- The many to many model multiplexes many user threads to a smaller or equal number of kernel threads. This means we create kernel threads to handle incoming user threads. No one does this even though it works pretty well and has no drawbacks except for multiplexing overhead.

4.4 Thread Libraries (Pthreads)

- A thread library gives a dev an API for creating and managing threads.
- Asynchronous threading happens when a parent creates a child thread, the parent and child execute at the same time. Each thread operates independently, and the parent doesn't have to know when the child dies. These usually do not share data.
- Synchronous threading occurs when the parent creates one or more children and then waits for all of its kids to terminate before it executes, like fork join. These share data.
- Pthreads is a POSIX API for thread creation and synchronization. It is a specification for thread behavior, not an implementation. OS designers implement Pthreads however they want, as long as they adhere to the specs.
- The C library for Pthreads is pthread.h. A pthread thread is defined as pthread_t, with a set of thread attributes pthread_attr_t. Every pthread needs a set of attributes to run, and these are loaded with pthread_attr_init(&attribute var).
- To create a thread, use the function pthread_create(address of thread var, address of attributes, function to run, parameters). To close the threads and join them back up with the parent, use pthread_join(threads).
- To create a runner process, just make a void *nameOfRunnerProcess(void *params).

4.6 Threading Issues

- If a thread calls fork(), it can do one of two things. It can fork all of the threads into a new process, or it can just fork the one thread into a new process, there are 2 types of calls to use for this. If a thread calls exec(), it could replace all threads with a new process, ruining the app. So, the way to solve this is, forks with exec after them copy one thread, while forks with no exec copy all threads.
- A signal in UNIX notifies a process that a particular even has occurred. A signal will be generated by a event, delivered to a process, and handled by the process. Synchronous signals are sent to the process that causes it. Async signals are generated by events external to a running process. Default and user defined signal handlers exist. The kernel runs signal handlers based on the signal.
- To deliver a signal to a mono proc, just deliver it synch, easy. But a multi can have the signal delivered to the thread which it applies to, every thread, certain threads, or an assigned thread that get signals. The UNIX function for delivering a signal is kill(pid_t, int signal). However, POSIX has a pthread_kill(pthread_t, int signal) that does the same.
- Thread cancellation involves killing threads before they complete. A thread to be canceled is a target thread. Async cancelling is done by one thread killing the target, deferred cancelling is done by the target thread checking whether it should die in an orderly fashion. Pthread_cancel(tid) kills a thread. Default kills are deferred. A cancellation point is the point in which a thread can die orderly. A clenaup handler is invoked upon a pending kill request.
- Thread local storage is data assigned to a thread, like static java variables to a class.
- An intermediate data structure is needed between a user and kernel thread, known as a LWP lightweight process. APIs will assign LWPs to user threads, and allows for communication between kernel and user. The kernel notifies the LWP with upcalls, and upcall handlers handle the upcall and communicate it to the user.

PROCESS SCHEDULING (NEED TO FINISH)

- Cooperating Process – One that can be or can affect other processes in the system that are executing. Threads handle these when a logical address space is shared, but when sharing files and messages, something else is sued.

5.1 Background

- Concurrency creates issues. If you are filling a shared buffer, and edit the variable keeping track of the number of objects in the buffer in two programs, a processor can mangle the value of a shared variable.
- A race condition is one where several processes access shared data, and the outcome of the execution depends on the order in which access takes place.
- To prevent this, we need to make sure that only one program can edit shared data at one time.
- The act of preventing these problems is called process synchronization.

5.2 The Critical Section Problem

- A critical section is a segment of code in a process that can be changing shared variables, writing to files and other things. When one process is executing a critical section, no other process is allowed to execute their critical sections.
- The critical section problem is to design a protocol so these processes can communicate.
- The entry section is the section of code that requests entry into the critical section.
- The exit section gives control to other critical sections in other processes, and starts execution of the remainder section in the code.
- A solution to the CSP must satisfy these requirements:
    - Mutual exclusion – If a process is executing in its CS, no other processes can execute their CS.
    - Progress – If no process is executing in its CS, and some processes want to, the selection cannot be postponed indefinitely and those processes that are not executing in their remainder section can participating who can execute their CS.
    - Bounded waiting – A bound exists on the number of times that other processes are allowed to enter their CS after they make a request to enter their CS.
- Preemptive kernels allow a process to be preeempted, a nonpreemptive kernel does not. The non-kernel will have kernel processes run until they want to stop. Pre kernels allow for process synchronization at the kernel level.

5.4 Synchronization Hardware

- Locking is the premise of protecting critical sections through the use of locks, which can grow to be quite sophisticated.
- One way to prevent a critical section from being modified is to use a single core processor, and prevent interrupts when a shared resource is being accessed.
- This is not feasible in a multicore system. It is too time consuming to disable interrupts on all the cores.
- Many computers allow for swapping the content of two words atomically, or as one uninterruptable unit.
- Two functions of atomic modification of a word are described by processes test_and_set and compare_and_swap. In testandset, we have a lock set to false initially. Test and set is called to unlock the lock by setting it to true, and the critical section is then run. The lock is then set to false again. This way, another process cannot run until the first process that got to test and set first runs and resets the lock. Compare is similar, but uses expected values and comparison.

5.5 Mutex Locks

- Hardware based solutions bite dick. They aren't super important, and are inaccessible to app programmers. OS designers build software tools to solve the critical section problem. The mutex lock is the simplest tool, and stands for mutual exclusion.
- A mutex lock protects critical regions. It has two functions, acquire() and release(). Acquire is called before a critical section, and uses a shared boolean variable called available that indicates if the value if available (true) or not (false). It waits if available is false, and when available becomes true from another process, it makes available false again and starts the critical section. The release function just sets available to true and stops another instance of the mutex from waiting in another process.
- This has a disadvantage, busy waiting. This is called a spinlock, as the process must spin and wait for a lock to become open. Since, in a single core CPU, we swap between processes quickly, this can waste time if a lock takes a long time to open. This works well when locks are only used for a short time, as on a multiprocessor system we can devote a core to spinning.

5.6 Semaphores

- A semaphore S is an integer variable that, apart from initialization, is accessed only through 2 atomic operations, wait and signal. Wait is defined as:

```
Wait(S){
    While(S<=0){
        ; //busy wait
        }
    S--;
    }
```

- Signal is defined as:

```
Signal(S){
    S++
    }
```

- When one process modifies the semaphore value, no other process can simultaneously modify the same semaphore value. The same goes for accessing S in wait.
- A counting semaphore can range over an unrestricted domain, and a binary semaphore can range between 0 and 1, like a mutex lock.
- A counting semaphore can be used to control access to a resource with a finite number of instances. We initialize the semaphore to the number of instances. When the resource is used, we perform a wait operation on it, and signal the semaphore when done. This way, we let the semaphore know that the resource is being used, and in other processes, when the semaphore <=0, we can wait until the resource is available. It kills two birds with one stone.
- Semaphores can also be used to sync up processes. For example, we have 2 concurrent processes. We have S1 in the first process, S2 in the second, and we want S2 to run after S1. We signal after S1 is completed, and wait before S2 is completed.
- Our current wait and signal operation suffer from spinlock still, however. To fix this, we use a cool ass list implementation:

```
//Assume the semaphore variable has an int value and a list of processes.

Wait(semaphore *S) {
    S.Value()--;
    If(S.Value() <0){
        Add this process to S.List();
        Block();
    }
}

Signal(semaphore *S){
    S.Value()++
    If(S.Value() <=0){
        Remove a process P from S.List();
        Wakeup(P);
    }
}
```

- Block suspends the process that invokes it, it won't do jack shit. Wakeup makes it do shit again.
- If a semaphore.value() is negative, its magnitude is actually the size of the linked list! The weird fuckin' way I think about it is this: Imagine there's a bunch of kids waiting for baseball bats. There's like 20 kids and 5 bats. The first 5 kids get to the bats, and bam, the semaphore value is now 0 as we're out of bats, and the rest of the poor fucks gotta wait for bat time. So the semaphore value is now the size of the line of kids waiting for bats, or –15. That line is the list.
- The list of waiting processes can be done with a link field in each process control block. Each semaphore can point to a list of PCBs.
- Semaphore operations HAVE to be executed atomically, which in actual English means that no two processes can modify them at one time.
- Deadlock occurs when two or more processes executing are waiting for an event that can only be caused by a waiting process. Remember to always remove things from the semaphore list in a LIFO order.
- Starvation/indefinite blocking occurs when a process is never removed from the queue it waits in.
- Priority inversion occurs when systems have more than two priorities on processes. It occurs when a low priority process is using a resource that a higher priority process wants to use. A chain of intermediate processes between the highest and lowest priority can make a high priority process wait for ages.
- One way to solve this is to make the lowest level priority process have the same priority as the highest

priority process wishing to access the shared resource. This is called priority inheritance protocol.

5.7.1 The Bounded Buffer Problem

- The producer and consumer processes editing the buffer share 4 data structures, int n, and three semaphores: mutex = 1 empty = n, full = 0. The pool consists of n buffers each capable of holding one item.
- The producer process will wait until the producer process has 1 open buffer. It will then wait for the mutex to reach 1, so it can add another thing to the buffer. It then signals the mutex and signals full, adding to the filled amount of buffers.
- The consumer process waits on the full semaphore has 1 or more filled buffer. It then waits on the mutex so as to not edit anything important. It consumes one item, then signals the mutex, then empty.

5.7.2 The Readers Writers Problem

- In the reader-writers problem, a data set is shared among a number of concurrent processes, with readers only reading the data set, not performing any updates, and writers both reading and writing the file.
- Only a single writer can access the data at the same time, and multiple readers can read it at the same time.
- The shared data is:
  - A data set like a file or array
  - A semaphore rw_mutex initialized to 1, whose purpose is to ensure that nothing is reading before we start writing
  - A semaphore mutex initalized to 1, whose purpose is to ensure that only one reader at a time increments the reader count
  - Integer read_count initialized to 0, which is used to keep track of the number of readers.
- The structure of a writer process waits on the rw_mutex, the writing is performed, and then we signal the rw_mutex.
- The structure of a reader process:
  - Initially we wait on the mutex, to not mess up the shared reader count data, and ensure that we have exclusive access to the critical data.
  - It increments read count, and if read count is the first reader, it waits on rw_mutex as a writer may be currently writing, and this would be the first reader and it may try to read while writing is happening.
  - We then signal mutex, so other readers can be created as we waited for the rw_mutex so we KNOW the writer is done. This also lets them start editing reader count.
  - Reading of the file is then performed.
  - After reading, we then wait on mutex, so as to not screw up the read count again.
  - After waiting, we decrement the read counter, and if there are 0 readers, we signal the rw_mutex. Then we signal the mutex at the end of any reader process. This will let the writer start writing again if there are 0 readers, and allow other reader processes to decrement the reader count.
- Starvation of writers can occur if you have infinite readers, meaning a file could never be written to.

5.7.3 The Dining Philosophers Problem

- The problem has five philosophers thinking and eating only.

-

Figure 5.13 The situation of the dining philosophers.

- The philosphers must have both chopsticks in hand to eat. It they do not eat, they think. You cannot just put a semaphore on each chopstick, as each dude could just pick up the one to this left and wait forever for the one on his right, causing a deadlock.

5.8 Monitors

- Using a semaphore can be handy, but it can be misused easily. We try to fix this with a new type of high level language construct called a **monitor**.
- An abstract data type is a construct that combines data with a set of functions that operate on that data. A monitor type is an ADT that has a set of functions that ensure mutual exclusion within the monitor. Only one process at a time can be active within the monitor. We have to define additional mechanisms within the monitor even with this. We can make condition variables that can be waited on and signaled. When we wait a condition, the process invoking the operation has to wait until another process signals that process specifically.
- Basically, think about the monitor like an object that controls access to something, and has functions that work on the data that is critical.
- The solution to the dining philosophers problem is below:

```
monitor DiningPhilosophers
{
   enum {THINKING, HUNGRY, EATING} state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
       (state[i] == HUNGRY) &&
       (state[(i + 1) % 5] != EATING)) {
          state[i] = EATING;
          self[i].signal();
      }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
    }
}
```

- Basically, the guy has to first get access to the monitor, then check the status of the guys next to him to eat. Since the monitor only allows one dude to eat at a time, no one picks up one chopstick all at the same time. So a monitor can limit one person to access the functions within it at one time, and it can also check conditions like if other people using it have done something.

CPU SCHEDULING

6.1 Basic Concepts

- In OS that have threads, kernel level threads are the things being scheduled by the OS. Process scheduling refers to general scheduling concepts, and thread scheduling refers to thread specific ideas.
- The objective of multiprogramming on a single core is to have a process running at all time to maximize processor efficiency.
- Process execution consists of a **cycle** of CPU execution and I/O wait. Process execution begins with a **CPU Burst, followed by an I/O Burst**. These are the periods that CPU or I/O operations are performed, respectively.
- CPU burst duration is typically short, with 2 millisecond burst durations being the most frequent.

IO bounds usually have many short CPU bursts, CPU bounds usually have a couple long CPU bursts.

- When A CPU get idled, the OS must choose a process in the ready queue to be executed by the **short term scheduler**, or CPU scheduler. It grabs a process in memory (ready queue (FIFO/priority queue/tree/unordered linked list based)), and throws it onto the CPU.
- We make decisions about scheduling the CPU during:
  - When a process witches from running to wait state.
  - When a process switches from running to ready state.
  - When a process switches from waiting to ready state.
  - When a process terminates.
- There's no choice in 1 or 4, we gotta schedule something new. Schemes that ONLY schedule during ¼ and ignore ¾ are called **nonpreemptive or cooperative**.
- When we do use 3 and 4 in our scheduling decisions, we have **preemptive scheduling**.
- If a piece of code does sensitive work on the kernel/OS, we need to block interruption, as concurrent modification of kernel data is disastrous. It is good to keep these sections of code short.
- The **dispatcher** is the module in the OS that gives control to the CPU of the processes selected by the short-term scheduler. It does this:
  - Switches context
  - Switches to user mode
  - Jumps to the proper location in the user program to restart the program.

  The time it takes the dispatcher to stop one process and start another is the **dispatch latency**.

6.2 Scheduling Criteria

- Different CPU scheduling algorithms (ways to choose things from the goddamn ready queue) favor one class of processes over another for selection initially. We use different characteristics of the algorithms to compare them:
  - CPU Utilization – We judge the algorithms on which keep the CPU as busy as possible. *Maximize.*
  - Throughput – We judge the algorithms on the number of processes completed per time unit (**throughput**). *Maximize.*
  - Turnaround time – Judge based on the time it takes from the submission of a process to the completion of the process, the sum of waiting to get into memory/ready queue, waiting in the fucking queue, executing on the CPU, and doing I/O. *Minimize.*
  - Waiting time – Judge based on the average time an algorithm waits in the ready queue. *Minimize.*
  - Response time – Judge based on the time from the submission request until the time of the first response to the user. *Minimize.*

6.3 Scheduling Algorithms

- The **First Come, First Serve** algorithm allocates the CPU to the process that requests the CPU first. This is done with a FIFO queue. We enqueuer PCBs and remove the oldest PCB from the queue upon a free CPU. This makes the waiting time very, very long.
- The **Shortest Job First** algorithm will associate the length of the processes CPU next burst with the process it belongs to. The CPU is assigned to the smallest next CPU burst. It is provably optimal, providing minimal average waiting time. It is difficult, though, is knowing the length of the next CPU burst for a process. For job scheduling, we use the process time limit when a user submits a job. It cannot be implemented for short term CPU scheduling however. For short term, we have to use the exponential average of the processes previous CPU bursts to estimate the next CPU burst length.
- **Nonpreemptive SJF** does not allow a process to be interrupted when it starts, but preemptive SJF does. This means as processes arrive in the ready queue, if its estimated burst is shorter than the currently running process, this process is placed in the CPU.
- **Priority Scheduling** associates a priority with each process, and the CPU is allocated to the process with the highest priority. Low integers means high priority and vice versa. In the preemptive version, when a higher priority process enters the waiting queue, it is put in the CPU. A process is blocked if it is ready to run but waiting for the CPU. Processes can be infinitely blocked if they are really low priority and high priority items keep coming into the queue in a process called **starvation**. In a technique called **aging**, a low priority process has its priority increased the longer

it waits.
- **Round Robin RR Scheduling** is designed for time sharing systems, and is similar to FCFS but has preemption. A small unit of time in RR is called a **time quantum**, usually 10-100 ms. The algorithm will treat the FCFS ready queue the processes arrive in as a circular queue, and then let each run for the specified time quantum. So, if p1,p2,p3,p4 are in the queue, and each takes 10 seconds, the order of execution would be p1,p2,p3,p4,p1,p2,p3,p4.

6.4 Thread Scheduling

- On OSs with user level threads, the os schedules threads instead of processes.
- On many to one and many to many models, the thread library schedules user level threads on an available LWP. This is known as **process contention scope**. This is called this way because competition for the CPU takes place between threads running in the same process. Usually done according to priority.
- To decide which kernel level thread to schedule on a CPU, the kernel uses **system contention scope**, in which all threads compete for the CPU. One to one models schedule threads only using SCS.

DEADLOCK (NEED TO FINISH)

- A **deadlock** is the event where multiple processes are requesting a finite amount of resources, and some processes have to wait to access the resources. In this event, if the other processes also wait for the same resources, some processes can never change state.

7.1 System Model

- In our system model for deadlock, a system consists of a finite number of resources to be given to a number of processes who all want those resources. The resources can be I/O devices, files, CPU cycles, or other things. The number of types of these resources available is the number of **instances of that resource**.
- When a process wants a resource, any instance of that resource should satisfy that want.
- Mutexes and semaphores are also considered system resources, and are a common source of deadlock. Since locks are usually designated to one data structure, we assign it to a resource class based on that data structure and the fact that it's a lock.
- A process has to request a resource before use, and release it after use. It follows this sequence:
  - **Request –** Request a resource. If available, grant to the process immediately, if not, process waits.
  - **Use** - Process uses the resource.
  - **Release** - Give up the resource. Make available to other processes.

7.2 Deadlock Characterization

- A deadlock can occur if these four conditions hold simultaneously in a system:
  - **Mutual Exclusion** - At least one resource is held in a nonsharable mode, meaning that only one process at a time can use the resource.
  - **Hold and Wait** - A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
  - **No preemption** - Resources cannot be preempted, meaning that a resource can only be released voluntarily by the process holding it after the process completes its task.
  - **Circular Wait** - A set of processes exist such that the first process is waiting for a resource held by the next one, and the last process is waiting for a resource held by the first one.

- Deadlocks can be described by a **system resource allocation graph**. This graph is a set of nodes V and edges E, with V having nodes of type P (processes) and R (resources). An edge goes from P to R if process P is requesting resource R. This is called a **request edge**. An edge from R to P exists if a process P is using R. This is called an **assignment edge**. P's are circles, R's are rectangles, and if there are more than one instance of a resource, the rectangle has a dot in it to represent that instance.
-

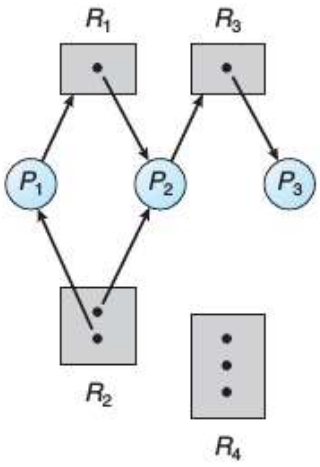
- 

Figure 7.1 Resource-allocation graph.

- If a graph contains no cycles, then no process in the system is deadlocked. If a graph contains a cycle, then a deadlock **MAY** exist.
- If the cycle involves a set of resources, each of which having only a single instance/dot, then deadlock exists. If they have multiple instances, deadlock can exist, but it also does not have to.

7.3 Methods for Handling Deadlocks

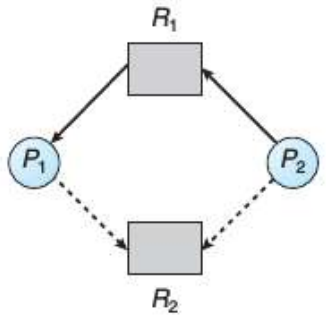
- There are three ways we can deal with the deadlock problem:
  - Use a protocol to prevent/avoid deadlocks, ensuring the system never enters a deadlocked state.
  - Allow the system to enter deadlock, but provide avenues for recovery
  - Ignore deadlock altogether, and pretend they never happen. This is what Linux and Windows do, making the developer avoid deadlocks.
- **Deadlock Prevention** keeps a system from *ever* deadlocking. This is done using a set of methods to ensure that at least one of the necessary conditions (the 4 mentioned earlier) cannot occur. This is done by controlling how requests for resources are made.
- **Deadlock Avoidance** requires the OS to be given additional info regarding the resources a process will want to use during its runtime. The OS will consider the requested resources, the resources it has available, and makes a decision whether the process has to wait or not.

7.4 Deadlock Prevention

- To prevent a deadlock, we have to ENSURE that one of the 4 conditions for deadlock cannot hold.
- We have to admit that the mutual exclusion condition has to hold. There is no way to prevent this, as every system has resources that are not shareable. Thus, at least one resource is always nonsharable.
- To ensure the hold and wait condition never holds (H E H E) is to make sure that a process does not hold any other resources when requesting a resource. The graph translation of this is that every process node has indegree zero when its outdegree is one or more. One way to do this is for a program to have all of its resources allocated before execution. Another way is to allow a process to request resources only when it has none. This kinda sucks, as we make resources wait while not being used, and starvation can occur since another process can use a resource forever.
- To defeat the no preemption condition for deadlock, we have to ensure that we can take resources away from a process before it is done with them. We can do this by using a protocol that takes all resources away from a process when it request a new one, and makes that process request those taken away resources along with its new one. This protocol is actually used, and is done with resources whose state can be easily saved and restored later like memory and CPU registers. It cannot be done with mutexes/semaphores.
- To ensure that the circular wait condition never occurs, we assign each resource a number, and make sure each process requests resources in an increasing order of the resource ID. This means the resource can request any resource if it hasn't requested one before, but afterwards it gets that resource, it can only request resources with IDs higher than the one it has at the moment.

7.5 Deadlock Avoidance

- This is using information in the scheduling of processes given to the scheduler by the processes to determine if certain processes should wait for other processes to finish before getting the resources they want.
- The algorithms that use this approach differ in the amount and type of info needed. The simplest/most useful model has each process declare the max number of each resource type it needs. The algorithm uses this info from all processes to ensure that a circular wait condition will never exist.
- We will look at two deadlock avoidance algorithms, safe state and the bankers algorithm.
- In Safe State, when a process requests an available resource, the algorithm has to decide whether immediately handing over the resource will leave the system in a safe state.
- A **safe state** is defined as the existence of a sequence of ALL process nodes such that for each P node, the resources that the P node can still request can be satisfied with the currently unallocated resources AND the resources held by the P nodes that come before it.
- This means that the if the P node needs a resource that's not immediately ready, it can wait until all the other P nodes are done with the resource, get the resource, then free it so subsequent P nodes can use the resource.
- A system can only deadlock when it is not in a safe state, so we want to avoid an unsafe state.
- For single instance of all resource type problems, we use a resource allocation graph, with a new type of edge called a **claim edge**. This edge means that a process may request the resource it points to from a process in the future. It is represented by a dashed line. When a process actually requests a resource, the line fills in. When a resource is released, the assignment edge turns into a claim edge.
-
-



7 Resource-allocation graph for deadlock avoidance.

- Resources must be claimed **a priori,** meaning that before a process executes, all of its claim edges must already appear in the graph. We usually only add claim edges when all of a processes edges are claim edges.
- We only grant a request edge when the request edge, if transformed into an assignment edge, would not form a cycle with all the claim edges in the graph. If a cycle is found, we know an unsafe state would be created upon creating that assignment edge.
- For example, P1 can request R2, but P2 cannot, because that would create a cycle. More in depth, if P2 got an assignment edge, then a cycle would be made if P1 requested R2, causing P1 to never release R1, causing P2 to never release R2, and so on and so forth. Deadlock.
- The Banker's Algorithm operates as follows:
    - When a process enters the system, it has to declare the max number of instances of each resource type it needs (less than the total number in the system).
    - Upon request of resources, the system decides if an unsafe state is created or not.

- Basically, a wait for graph takes the resource allocation graph, and removes the resources. Thus, an arrow from one node points to another node if it is waiting on that node to release a resource it needs.

-

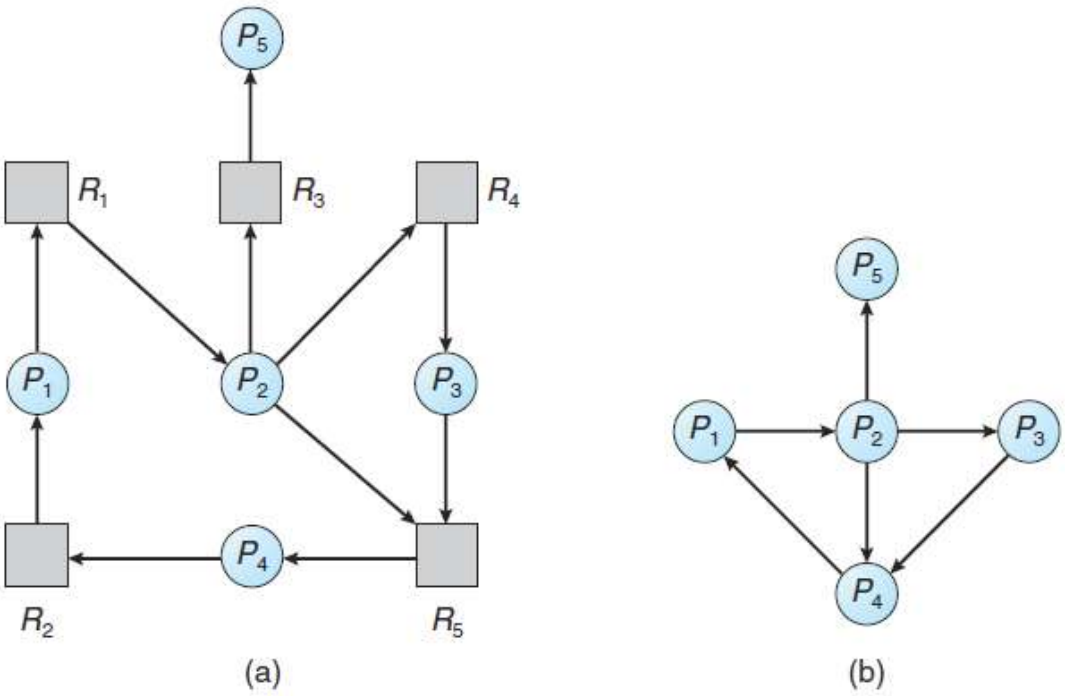


Figure 7.9 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

MAIN MEMORY

- To efficiently schedule processes, we have to store processes in memory in the ready queue. This means we have to know how to allocate memory correctly.

8.1 Background

- **Memory** consists of a large array of bytes, each with its own address. This array is central to the operation of a modern OS. Instructions are fetched by the CPU according to the value of the program counter on the CPU.
- The only general purpose storage a CPU can access directly is the main memory and the registers in the processor. All instructions have to be moved to memory before a CPU can execute them.
- One tick of the CPU clock is usually all it takes to perform a simple operation with a CPU register, but an operation using main memory can take many cycles of the CPU clock, causing a **stall** since data needs to come in via BUS.
- To remedy the stall, we use fast memory built into the processor chip itself, called a **cache**. This allows for faster memory usage per process.
- We use memory to make sure that the user level processes never touch kernel level processes, and that each process has a separate space in memory. This allows protection between all kinds of processes.
- To provide this protection, we use a **base register** and a **limit register**. The base holds the smallest legal memory address, and the limit specifies the size of the range, or the amount of addresses from the base register the memory space can use.
-

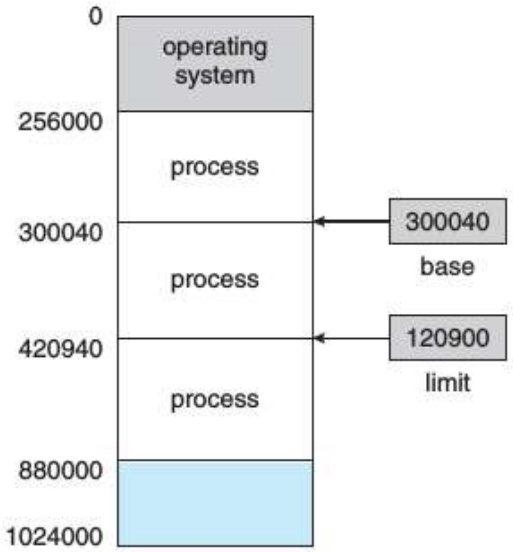


Figure 8.1 A base and a limit register define a logical address space.

- To protect memory space, every address generated in user mode is compared in the registers with memory, and if it is outside of its allowed limit and base, it is fatal errored out.
- A program resides on a disk as a binary executable file. To execute this file, we bring it into memory and place it in a process. Processes on disk waiting to be brought into memory wait in the **input queue**.
- Most user processes can reside in any part of the RAM. Programs written by the user are usually represented in different ways before the reside on RAM however. Addresses can be represented much differently, with a symbolic address being a programming language variable. The process of **binding** assigns these symbolic addresses to relocatable addresses to be used in memory allocation, which is then read by the linkage editor to assign a real address in RAM.
- The binding of instructions and data to memory can be done at any step along the way from source code to execution:
  - **Binding At Compile Time** allows for **absolute code** to be generated, and can only be done when the location of where a process will reside in memory is known at compile time.
  - **Binding at Load Time** allows for binding when compile time binding is impossible. This means binding is not done at compile time, but when the program is loaded into memory from disk. A compiler has to generate **relocatable code**, meaning that if the starting address changes, we just have to reload the user code to change the values based off the starting base address.
  - **Binding At Execution Time** allows for binding when a program is loaded from memory into a process.

- An address generated by the CPU is a **logical address**, where an address seen in RAM, loaded into the **memory address register** is referred to as a **physical address**. Compile/Load time binding methods generate identical logical/physical addresses, but execution time results in different ones. In this case, we call a logical address a **virtual address**.

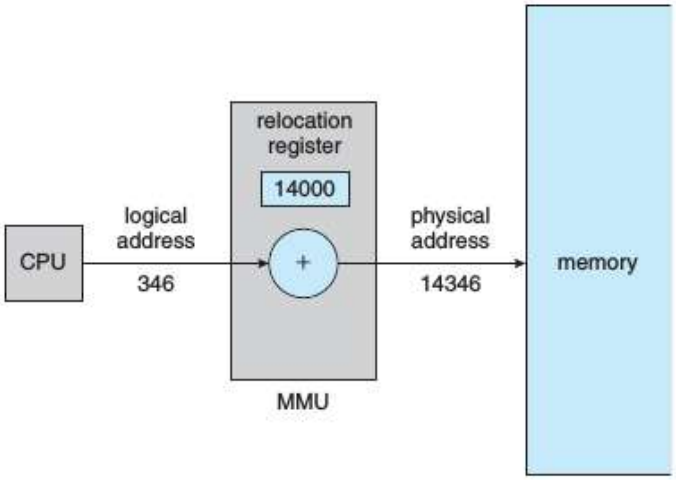
- 

Figure 8.4 Dynamic relocation using a relocation register.

- The mapping of virtual to physical address space is done by a **memory management unit (MMU)**. The base register for the mapping of a logical address to a physical address is located in the **relocation register**. In a basic scheme, the logical address is added to the relocation register and then put in physical memory. In user programs, we only worry about the maximum allowed virtual address, and always regard our base address in memory as 0.
- **Dynamic Loading** does not allow for ALL needed routines in a user program to be loaded into memory, but only when they are called by the main process.
- **Dynamic Linking** is performed at execution time, and it is used in a user program by loading system and library code off the disk when called in a user process. A **stub** is a way to locate these libraries.

8.5 Paging

- **Paging** is a memory management scheme that allows for the physical address space of a process to be noncontiguous. It also avoids external fragmentation.
- To do paging, basically we break physical memory down into fixed size blocks called frames, and logical memory into fixed size blocks called pages. Upon execution, pages are loaded into any available frames from the backing store or the file system. The backing store is broken down into fixed size blocks that are the same size as one or a multiple of the frame size.
- Every address generated by the CPU is divided into a **page number p and a page offset d.** The page table uses the page number as an index, and at that index contains the frame number/actual physical address, and the offset is used from that address to find what the CPU wants.

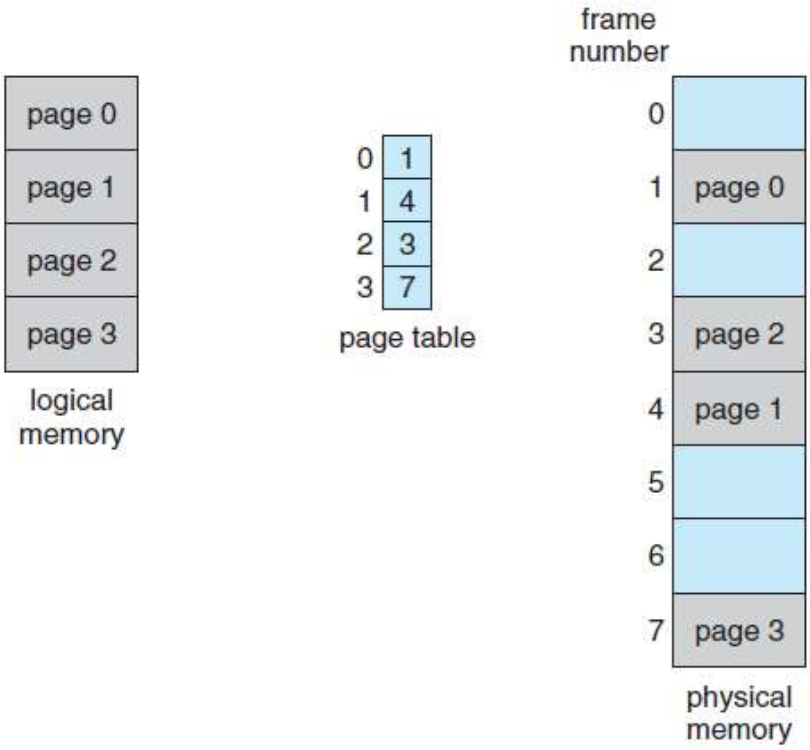


Figure 8.11 Paging model of logical and physical memory.

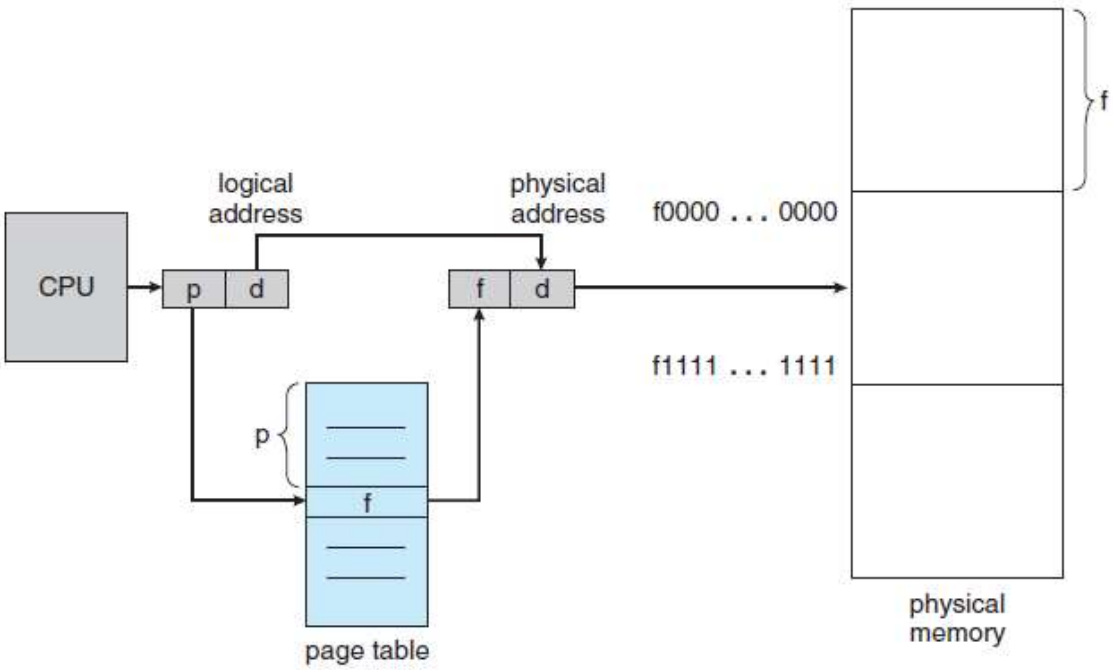


Figure 8.10 Paging hardware.

- Page size is defined by hardware. If the size of the logical address space is 2^m, and a page size is 2 ^n bytes, then the m-n higher (more significant) bits of a logical address give the page number, and the n lower (less significant) bits give the offset.
- On average, internal fragmentation in paging takes up 1/2 of a frame.

- When a process arrives to be executed, it gets its size expressed in pages, so the paging hardware has to determine if there are enough frames for the processes pages.
- With this info, we now know an OS has to keep track of frames as well, and we do this with a **frame table**. This has one entry per each frame, with an allocated or unallocated status. If it is allocated, it indicates what page it is given to.
- In hardware, the page table is kept in main memory, with a **page table base register** pointing to it, and a **page table length register** indicating the size of the page table. Every data/instruction access in memory requires two memory accesses, for the table and data, making things slow.
- To solve this, we use a **translation look aside buffer**, which is high speed and has no delay in finding things in memory. It is a cache on the CPU, and is between 32 and 1024 entries. A CPU will generate a logical address, and then it will look up that address in the TLB. If it is there, it immediately grabs the data, if not, we have to check the page table as a **TLB Miss** has happened. The page number and frame number will be added to the TLB after a miss, and we still grab the data, although slower. An entry is **wired down** when it is forever stored in the TLB.
- Some TLBs use **address space IDs**, which uniquely ID each process, and provides address space protection. If the ASID does not match the process ASID in the page table, we treat it as a TLB Miss. If we do not do this, we have to wipe the whole TLB between every context switch so we do not get the wrong addresses.
- The **hit ratio** is the amount of time the page number we want is in the TLB. To calculate the **effective memory-access time** of an instruction, we multiply the hit ratio by the TLB lookup time and add it to the inverse of the hit ratio multiplied by the page table lookup time.
- We can use an extra bit in the page table to indicate that a page should not be written to or read for protection. We can also have a **valid/invalid bit**, which tell us whether the page we are looking for is in the logical address space of a process.

8.6 Structure of the Page Table

- **Hierarchical Paging** is when a logical address maps to an outer page table, then an inner page table, then an actual frame. The logical address has the (in a 32 bit system) the first 10 bits ID the page number in the outer table, the next 10 bits ID the page number in the inner table, and the last 12 bits ID the offset from the beginning of the frame. This is called **forward mapped page table.**
- **Hashed Paging** is done for address spaces larger than 32 bits. Each element in the page table has the page number used by the logical memory, the physical frame number, and a pointer to the next element in the linked list. The idea is that the page number is gotten from a hash function, and the list is searched for a collision, and then the frame number is returned.
- An **Inverted Page Table** has one entry for each frame of memory, and assigns a process ID to each frame as well as a page number. This is used for lookups.

VIRTUAL MEMORY

9.1 Background

- **Virtual Memory** involves the separation of logical memory as perceived by users from physical memory. It lets programmers focus on only the virtual memory they use, not the physical memory. Virtual memory can be larger than physical memory as well.
- **Virtual Address Space** belongs to a process, and it refers to the logical view of how a process is stored in memory. This starts at a certain logical address, and exists in contiguous memory. In reality, we know from the previous chapter that this virtual memory can be mapped to different frames in physical memory. The model for this address space is simple, it has the code at the bottom, then the global data, then the heap, then a hold/shared data, then the stack. The traditional view.
- Virtual address space that include holes are known as sparse address spaces. These holes can be filled by stack or heap data. They are also used for dynamic linking using shared libraries.

9.2 Demand Paging

- **Demand Paging** involves only loading pages as they are needed by the user process. Pages are only loaded when demanded by the program execution.
- This system is similar to page swapping, where a process will reside in secondary memory. When we want to execute a process portion, we use a **lazy swapper** that only swaps a page into memory as it is needed. However, as a swapper is usually known as swapping an entire process into

memory, we will call a lazy swapper a pager from hereon, as a pager is concerned with only the individual pages.

- When a process is needing swapping in, the pager will guess which pages will be needed. We need a way to distinguish the pages that are in memory, and the pages that are in secondary. We can use the valid/invalid bit way from CH 8, which is to set a bit to valid in the page table if the page is legal and in memory, and a bit is set to invalid if a page is either not in the logical address space of the process, or if it is on the secondary memory.
- If a page is invalid, we do not have to worry about it if we never have to access the page. If the process only needs pages that are memory resident, or in memory, we never find an issue since we never access an invalid bit.
- If we do access an invalid page, we cause a **page fault**, which sends a trap to the OS that brings the desired page into memory. The process goes as follows:
    - Check an internal table usually in the PCB for the process to check if the reference was valid/invalid.
    - If it was invalid, we terminate the process like normal. But if it was valid and we do not have the page, we will have to page it in.
    - We find a free frame.
    - We schedule a disk operation that reads that page into the new allocated frame.
    - We modify the internal table kept with the PCB and the page table to show the new page is in memory.
    - We restart the instruction.
- **Pure Demand Paging** is when we never bring a page into memory until it is required.
- **Locality of reference** is the reasonable conclusion that demand paging will never result in multiple processes each having instructions needing several new pages of memory each. If this were to happen, demand paging would be impossible.
- Hardware needed to support demand paging is exactly the same as page swapping. You need a page table, and you need secondary memory, usually allocated **swap space** in a high speed secondary memory location.
- We need to be able to restart any instruction, which is a simple enough process. We just need to be able to restart in exactly the same place and state as when a page fault occurs.
- Demand paging can significantly affect the performance of a computer. The **effective access time,** the time it takes for memory to be accesses plus the time it takes to handle a page fault, tracks this. The formula is:
-
- $$\text{effective access time} = (1 - p) \times ma + p \times \text{page fault time}.$$
- P is the probability that a page fault will occur. Ma is the time it takes for the computer to access memory at a base. There are 3 major components of the page fault time, the page fault interrupt, the reading in of the page from memory, and the restart of the process. The first and last tasks can be reduced, to about 1 to 100 microseconds. The actual page-switch time can be probabilistically close to 8 ms. This time is also if the swap is the first thing in the waiting queue for I/O on the disk.
- All of this basically means that the effective access time is directly proportional to the page-fault rate. We need to keep page faults to a minimum to make demand paging actually useable. This usually entails keeping fewer than one memory access out of 399,990 to fault.

9.3 Copy on Write

- **Copy on Write** is a technique used by the UNIX fork() call that allows the parent and child process created by fork to share the same pages. When either tries to access pages from this shared space, the OS creates a copy of that page and that copy is edited instead.
- Vfork() does not use COW, but instead lets the child process use the same pages as its parent, and also allows it to edit the pages of its parent directly. This is usually only used when creating processes that will get a new address space.

9.4 Page Replacement

- **Over-allocation** is when in demand paging we allow more processes to run that have actual memory used that is less than the current available used in RAM, but have maximum memory used that is greater than the memory available in RAM.
- When over allocation happens and a page fault occurs, the OS will grab the needed page from disk, but see that there are no free frames available. From here, the OS can do many things. One

thing that can be done is the complete swapping out of a process and reducing multiprogramming. The most common solution is **page replacement**.

- Page replacement involves finding a frame that is not being used and free it. We write its contents to the swap space, and change the page table to show that the page is no longer in memory. The page replacement process now goes as follows:
  - Find the location of the desired page on the disk.
  - We find a free frame. If there is one, use it. If there isn't, use a page replacement algorithm to choose a **victim frame**, and write the victim to disk and change the page and frame tables accordingly.
  - Read the desired page into the new frame, change the page and frame tables
  - Continue the process from where the page fault happened.
- Since this has two page transfers, it can double the page fault service time and increase effective access time a lot.
- If we use a **modify/dirty bit**, we can reduce the double page fault service time overhead. Each page or frame has a modify bit associated with it in the hardware. The bit is set by the hardware any time the page has been modified. When we choose a victim page, if the modify bit has been set, we know changes have occurred to it and it must be written to disk. If it hasn't, the same info in the page is in the disk, so we can just replace it with the new page. This can reduce PFT by ½.
- To do demand paging well, we need a **frame allocation algorithm and a page replacement algorithm.** We have to decide how many frames we can allocate to each process, and how to select frames to be replaced when a new page is needed. We want to choose the PR algo with the lowest page fault rate. We begin with page allocation algorithms.

9.4.1 Page Replacement Algorithms

- To evaluate these, we run it on a particular string of memory references and compute the number of page faults. This string of memory references is called a **reference string**. To create these, we usually take a string of addresses from an OS, and only include the address spaces that are not contiguous in memory. This means that memory accesses that are close to each other are only counted once in the string, since these immediate accesses will not cause page faults. For example:
-

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

- At 100 bytes per page, this sequence is reduced to the following reference string:

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

- We can see that after the first time the 600 range space was referenced, the 100 range address space was referenced 4 times, but is only counted once in the reference string.
- If the number of frames is large, the amount of page faults will shrink as well.
- The FIFO algorithm associates with each page the time that the page was brought into memory. When a page has to be replaced, the oldest page is chosen. If a page is already in memory it isn't replaced in the queue that holds all the page frames, but if it isn't, It replaces the first one.
- Belady's Anomaly is when the number of page faults increases as the number of frames available increases. This occurs in the FIFO algorithm.
- The optimal page replacement algorithm is one that has the lowest page fault algorithm and does not suffer from Bealdy's anomaly.
- An algorithm has been found that does this called OPT/MIN. This algorithm replaces the page that is not used for the longest period of time. This is hard to implement, as we do not know the future of the reference string.
- If optimal page replacement is not feasible, then we use an approximation of it. The LRU method (least recently used) takes the page that has not been used for the longest period of time. This is a good page replacement algorithm, but implementing it is hard. We can use a counter, by associating each page table entry with a time of use field and a clock on the CPU that will update that field. We can use a stack as well, by removing a referenced page from the stack and put on top, and removing the bottom of the stack.

- LRU approximation needs to take place, as the hardware is specialized and slow.
- Page Buffering algorithms use a pool of free frames, and evict victims to add to the pool. It keeps a list of modified pages, which are written when the backing store is idle and freed into the pool, or keep the frame contents intact if they are referenced again.

9.5 Frame Allocation

- Since each process needs a minimum amount of frames, we need an algorithm that will give it that minimum or a little more. Two algorithms exist, the fixed allocation and the priority allocation.
- Fixed allocation gives equal allocation to each process. If there are 100 frames and 5 processes, they each get 20 frames without using a free frame buffer pool. If we proportionally allocate these instead of equal, we allocate this many frames:
  - $s_i = \text{size of process } p_i$
  - $S = \sum s_i$
  - $m = \text{total number of frames}$
  - $a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$
- This will dynamically change as process sizes and the amount of processes we are running change.
- Priority allocation uses proportional allocation using priorities. If a process generates a page fault, we select a replacement for one of its frames from a process with a lower priority number than its one.
- To select a frame to replace, we can do **global replacement** which takes a replacement frame from the set of all frames, but process execution varies a lot, and throughput increases.
- **Local replacement** has each process replace pages in frames from its own set of frames, which makes processes perform more consistently, but underutilizes memory.

9.6 Thrashing

- A process thrashes when it does not have enough pages, and has a high page rate. This means thrashing is when a process is always busy swapping pages in and out, spending more time on that than executing.
- We can limit thrashing with a local replacement algorithm. This way it doesn't make other processes thrash as well. It will still increase effective access time though, as they clog up the queue for page swapping.
- We use the locality model that states that as a process executes it moves from locality to locality. A locality is a set of pages that are actively used together. Programs usually use several different overlapping localities.
- Temporal locality is the value of probability that a variable will be accessed again in the near future. Spatial locality is the probability that data nearby other data will probably be accessed in the near future.
- The **working set model** is based on localities, and uses DELTA to define the working set window. The working set is the set of pages in the most recent DELTA. This sucks.
- A way better, less archaic and dumb way of doing this is **Page Fault Frequency**. We establish upper and lower bounds on the page fault rate for a process, and if the processes page faults exceed the upper bound, we give it another frame. If it grows lower, we take a frame away.

MASS STORAGE STRUCTURES

10.1 Overview of Mass-Storage Structure

- Magnetic Disks (HD) provide the bulk of secondary storage in PCs.
- Each disk platter has a flat circular shape, like a CD. The two faces are covered in a magnetic material, and we record info to the plates using magnetism. A read write head sits above each surface of each platter, and the heads attach to a disk arm that moves all the heads as a unit. Each platter is divided into circular tracks, which are divided into sectors. The set of tracks that are at one arm position make up a cylinder.
- Disk speed has two parts. The **transfer rate** is the rate at which data goes from the drive to the

computer, and the **positioning** time consists of two parts, the time it takes to get the needed sector with the head (rotational latency) and the time it takes for the disk arm to get to the right cylinder (seek time).
- When the head touches the platter, a head crash happens and ur shit is broke holmes 4eva.
- A disk drive is attached to a computer by a set of wires called an **I/O bus**. SATA, USB, and FC busses are common. Data transfers on a bus are carried out by special electronic processors called **controllers**. The host controller is the controller at the computer end of the bus, and the disk controller is built into the disk drive. To do disk I/O, the computer puts a command in the host controller, sends it to the disk controller, and the disk controller uses the disk drive hardware to do the command. Disk controllers have a cache, with data going from platters to cache, and from cache to host.
- Solid state SSDs are nonvolatile memory, faster, and die quicker than HDs with platters. Busses can slow these, so some connect over PCIs.
- **Magnetic Tape** has a slow access time, but can hold huge amounts of data. We use them to backup infrequently used info, and for transferring data.

10.2 Disk Structure
- Modern magnetic disk drives are addressed as large 1D arrays of logical blocks. These are usually 512 bytes. Sector 0 is the first sector of the first cylinder, and is mapped to the first logical block array index. The array continues from the outside track into the inside track, and then down a cylinder.
- We can in theory convert a logical block to a cylinder number, a track number, and a sector number. Defective sectors and variable sector numbers in tracks make this hard.
- The farther a track is from the disk center, that track has more sectors as the circumference of that track is larger. Outer holds 40 percent more sectors than inner.
- In **constant linear velocity,** the above holds, as the drive increases its rotation speed as the head moves inwards to maintain data speed. In **constant angular velocity**, the speed remains constant, and less data is stored per sector in the outer track than the inner track.

10.3 Disk Attachment
- Computers can access disk storage in 2 ways, via I/O ports (also called **host attached storage**), or **network attached storage**.
- Host attached storage includes HDs, RAID arrays, CD and DVD drives.
- A NAS drive is a storage system accessed remotely over a data network. You can access NAS via remote procedure calls in UNIX and Windows, and these are done using TCP or UDP over an IP network, which is usually the LAN. You can use this to share storage between machines.

10.4 Disk Scheduling
- One algorithm for disk scheduling is FCFS. The disk will read from the first block in the array, and the total time it takes for this algorithm is the amount between each block value in the queue.
- Shortest Seek Time First selects the request with the minimum seek time from the current head position. This is analogous to SJF scheduling. This reduces head movement greatly. Head movement is measured in cylinders.
- SCAN has the disk arm starting at one end of the disk, and it moves to the other end servicing requests until it gets to the other end of the disk, and then the direction reverses and it does the same thing. Sometimes called the elevator algorithm. If we have a uniformly dense queue of requests, then the largest density at the other end of the disk will wait the longest.
- CSCAN provides a more uniform wait time than SCAN. It is identical to SCAN, except when it reaches the other end of the disk, it just goes back to the other side. It treats the cylinders as a circular list that wraps around from the last cylinder to the first one.
- LOOK is a version of SCAN, CLOOK is the same for CSCAN, and the arm only goes as far as the last request in one direction and then reverses for LOOK, and the arm only goes as far as the last request in one direction and goes back to the other end of the disk for CLOOK.
- Choosing a disk scheduling algorithm can be hard. SSTF is common, but SCAN and CSCAN perform better for systems that place a heavy load on the disk. Performance depends on the number and types of requests.
- The disk scheduling algorithm should be written as a separate module of the OS so it can be replaced if needed later.

10.5 Disk Management

- Before a disk can store data, it has to be divided into sectors that the disk controller can use, in a process called **physical/low-level formatting**.
- This will fill each sector with a header, a data area, and a trailer data structure. The header and trailer have info for the controller, like the sector number and error correcting codes. These codes are used to label disk sectors as bad and corrupted. These are reported as soft errors.
- **Partitioning** separates the disk into one or more groups of cylinders, and OS can treat each cylinder as its own separate HD. **Logical formatting** is the act of creating a file system on the HD. **Clusters** are groups of blocks that the file system lumps together. DISK IO is done using blocks, but file system IO is done using clusters. **RAW DISK MODE** allows for direct app access to the HD.
- A **bootstrap** is the initial program that runs when the PC is booted up. This program initializes all aspects of the system. It does this by loading the OS kernel onto the memory from disk, and jumps to the first instructions for OS work. The bootstrap is stored in **ROM**, which is stored in a fixed location that the CPU always knows. Its job is only to start the full bootstrap program saved on the HD. This program is in a partition called the **boot disk**.
- **Bad blocks** are sectors of the disk that have become defective. We can either mark bad blocks when formatting to make sure the OS doesn't use them, or we can set aside a list of spare, just in case sectors in a process called **forwarding** to use when sectors corrupt. Every cylinder has a few spare sectors, and spare cylinders exist too.

FILE SYSTEM INTERFACE

11.1 File Concept

- A **file** is a logical storage unit. These are mapped by the OS onto physical devices. More specifically, it is a named collection of related info recorded on secondary storage. Files represent programs and data. Usually, a file is a sequence of bits, bytes, lines or records with meaning defined by the file creators user.
- A file has a couple of attributes:
  - Name, the human readable name.
  - Identifier – The ID the file system uses. The non human readable name.
  - Type – the type of file it is. Suffix.
  - Location – Pointer to the device and location of the file on the device.
  - Size – The size of the file in bytes or blocks.
  - Protection – Access control determining who can read write execute and other stuff
  - Time, date, user ID info
- Newer file systems support extended file attributes, like character encoding and checksums.
- To define a file properly, we need to consider the operations that can be done on it. Here are some basics:
  - We can create files, by finding a space in the file system, and then allocating space for it. We then record an entry for the file in the directory.
  - We can write a file by calling a syscall to write to it. This syscall searches the directory to find the file, and keeps a write pointer to the location in the file where the next write takes place. It is updated when we write.
  - To read a file, we find it like a write, and keep a read pointer where the next read will happen. Read pointers are updated upon reading. We can combine the two into a current file position pointer.
  - We can reposition in a file, meaning we move the pointer. This is known as a file seek.
  - We can delete the file by finding it in the directory, and release all file space and erase the directory entry.
  - Truncating a file does not delete it from the directory but frees all file space.
- To find files, the OS uses the **open-file table** to record info about all open files so we do not have to search repeatedly every time we do an operation, only when we open the file. Other tables include a per-process OF table that keeps track of a processes files and the info needed by the process, like the file pointers and the file mode. These then point to a system wide table that has all open files for all processes that contains info like file size, disk location, and others. The **open count** is a number that keeps track of how many processes have a file open. Upon reaching zero, it is removed from the table.
- OSs can lock an open file or a section of it. They are similar to reader writer locks. A **shared lock** is like a reader lock, as multiple processes can access a file/get the lock concurrently. An **exclusive**

lock is like the writer lock, only one process can use the file.

- Locks can be either **mandatory** or **advisory**. Mandatory locks prevent other locks from getting an exclusive lock, but advisory locks give you the lock but alert that another process is using it.
- A common technique for file types is using a file name and extension. The name is what we identify the file with, the extension tells the OS how to use the file based on type. Think cool.c.
- A **shell script** is a file containing commands to the OS.
- Files need to have structures that match the programs that use them's expectations. A good way is to have only one officially supported executable file type, and then have application programs interpret other file types.
- Disk systems usually have blocks defined by the size of sectors on the HD. Since logical records do not fit into one block, we will have to pack a number of records into multiple blocks. This means the file can be considered a sequence of blocks in the HD.

11.2 Access Methods

- Systems provide multiple ways to access information stored in files.
- **Sequential Access** is when file info is processed in order, one record after another. **Direct access/relative access** files are made up of fixed length logical records that allow programs to read/write records in no particular order quickly. The file is seen as a numbered sequence of blocks or records. Databases usually use this. We usually use a relative block number to find files, which is a block number relative to the block number of the file start. The file start is set to 0, and we allocate from there, but use the true block number when writing.
- You can build other methods on top of direct access. We can use an index to have pointers to various blocks. To find a record, we search the index and follow the pointer to the correct location.

11.3 Directory and Disk Structure

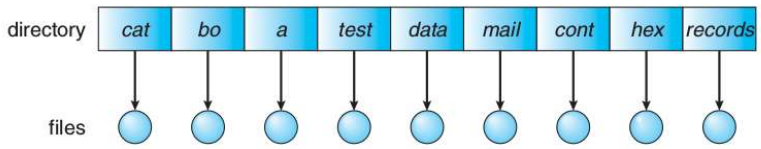
- A storage device can be **partitioned** into fractions of itself to hold different file systems. These fractions are typically called volumes, which can be a subset of the device, the whole device, or multiple devices linked together. Each volume has a file system that contains info about files in the system. These are kept in a device directory that records info such as name, location, size and type for files in the volume.
- The directory is like a table that translates file names into their directory entries. We can search for files based on name, create files, delete files, list a directory's contents, rename files and traverse the file system.
- The single level directory is just a single table, with every file in the OS being stored with unique names:
- 

Figure 11.9 Single-level directory.

- This kind of sucks. We would get too many files and organization would be a nightmare.
- The two level directory has a master file directory, that creates a subdirectory for each user in the MFD, called the user file directory. This way, every user gets their own single file directory. This still is unorganized, and makes it difficult to share files between UFDs. A **path** is a user name and then a file name in this structure. The full path usually uses the disk volume, then the user, then the filename. A search path is the list of directories that are searched when a file is named to be found.
- The tree directory allows users in their UFD to create multiple subdirectories of files. Since the path changes for each subdirectory, different subdirectories can have identical file names. When the OS is loaded and a user logs in, the OS's accounting file is searched for a pointer to the users initial directory, and they start there.
- Absolute path names start at the root of the directory (usually the volume) and go downwards to the file. Relative path names are started from the current directory.
- The acyclic graph file system structure is the same of the tree, except two directories can share subdirectories. This allows for multiple user directories to share files and subdirectories.

- In UNIX, we implement a shared directory by using a symbolic link, which is a pointer defined by an absolute/relative path that allows for the finding of the shared directory. This way, we can get to and from directories. The finding of this using the path is called resolving the link. When a link to a file or directory points to a deleted item, we just delete the link.

11.4 File System Mounting

- A file system has to be mounted before it is used by processes on the system. This means the directory structure has to be built out of multiple volumes.
- The **mount point** is the location where the file system is to be attached. The mount point is usually empty. Think of a mount point as the root of the directory tree.
- After the mount point is decided, the OS verifies that the device containing the file system contains a correct directory structure.

FILE SYSTEM IMPLEMENTATION

12.1 File System Structure

- To improve I/O efficiency, we transfer between disks and memory in units of **blocks**. A block can have one or many sectors within it.
- **File systems** provide efficient and convenient access to the disk by allowing data to be stored found and retrieved easily. The system itself is composed of many different levels:

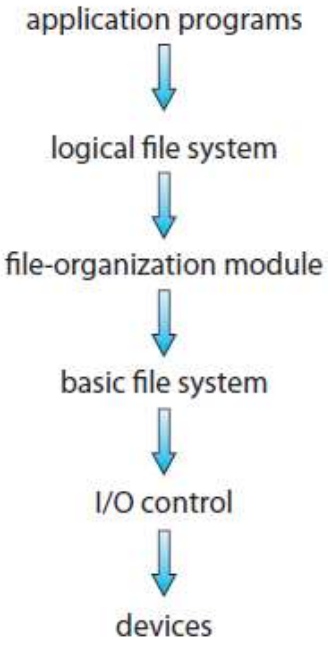


Figure 12.1 Layered file system.

- The I/O control level consists of device drivers and interrupt handlers that transfer info between main memory and the disk system. It takes in commands like "read block 123" and turns them into instructions for the actual device like a hard drive to follow.
- The basic file system needs to only issue generic commands to the appropriate device driver on the I/O control level to read and write blocks on the disk. (A block is identified by its cylinder, track and sector). This level also manages memory buffers/caches that hold various blocks of information. If the buffer is full, this layer will make space.
- The file organization module has knowledge on files and the logical blocks they belong to, as well as the physical blocks. This layer translates between the logical blocks from the layer above it to the physical blocks of the layer below it. In the logical blocks, every block is numbered 1 to N. These are translated to vastly different physical block numbers. This layer also manages free space.
- The logical file system manages metadata information, like the structures we learned in 11 as well

symbolic file names. A file control block/inode contain info about files, like their ownership, permissions and location of the contents.

- The layering of the file system responsibilities makes sure that code duplication is minimized. Multiple file systems can use the I/O control and even the basic file system sometimes.

12.2 File System Implementation

- Both on-disk and in-memory structures are needed to implement a file system. On the disk, the file system can contain info on how to boot an OS, the total # of blocks, the number and location of free blocks, the directory structure and individual files.
    - A **boot control block** can contain info needed by the system to boot an OS from that volume. It is usually the first block of a volume.
    - A **volume control block** contains volume/partition info like the number of blocks it has, and the number of free blocks. It also has a number of free FCBs.
    - A directory structure is stored on disk. In NTFS, its called the MFT master file table, in UFS it is the file names and their inode numbers.
    - Every FCB is stored in the disk as well, with the details on a file.
    -

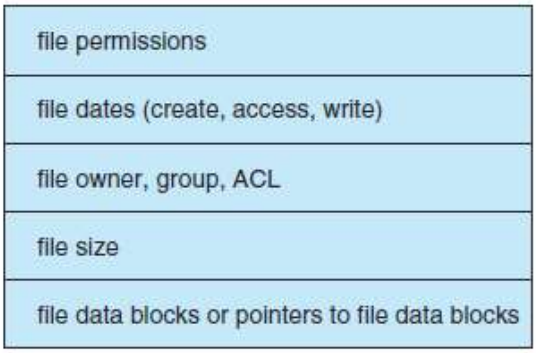
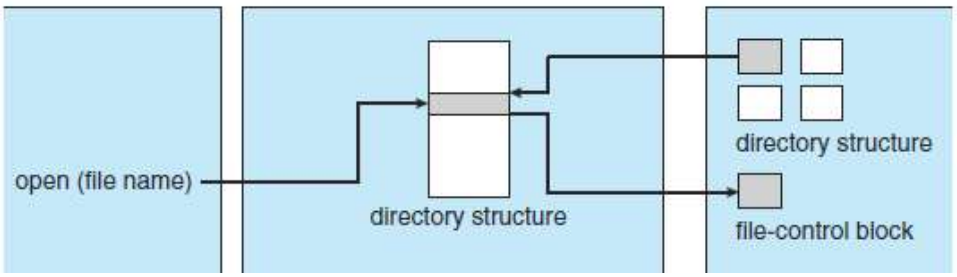


Figure 12.2 A typical file-control block.

- In memory information is used for both file system management and performance improvement via caching. This data is loaded at mount time, updated during file operations, and discarded at dismount. Some examples are:
    - A **mount table** is loaded that contains info about each mounted volume.
    - An in memory directory structure cache holds directory info of recently used directories.
    - The **system wide open file table** has a copy of each FCB that corresponds to an open file.
    - The **per process open file table** has a pointer to the appropriate entry in the system wide open file table.
    - Buffers hold file system blocks when they are transferring to/from the disk.
- In the process of making a new file, an app calls the logical file system, which allocates a new FCB. The correct directory is loaded and the filename and FCB are put into it, and this info is written to disk.
- The open() syscall passes a filename to the logical file system. The LFS searches the system wide open file table to see if the file is being used. If it is, a per process open file table for that file is updated to point at the system wide table.
- If not, the directory is searched for the file name, and when found the FCB is transferred into the system wide open file table. Then an entry is made into the per process open file table with a pointer to the entry in the system wide open file table. The open syscall will then return a pointer to the entry in the per process open file table, which is used by all file libraries in programming languages.



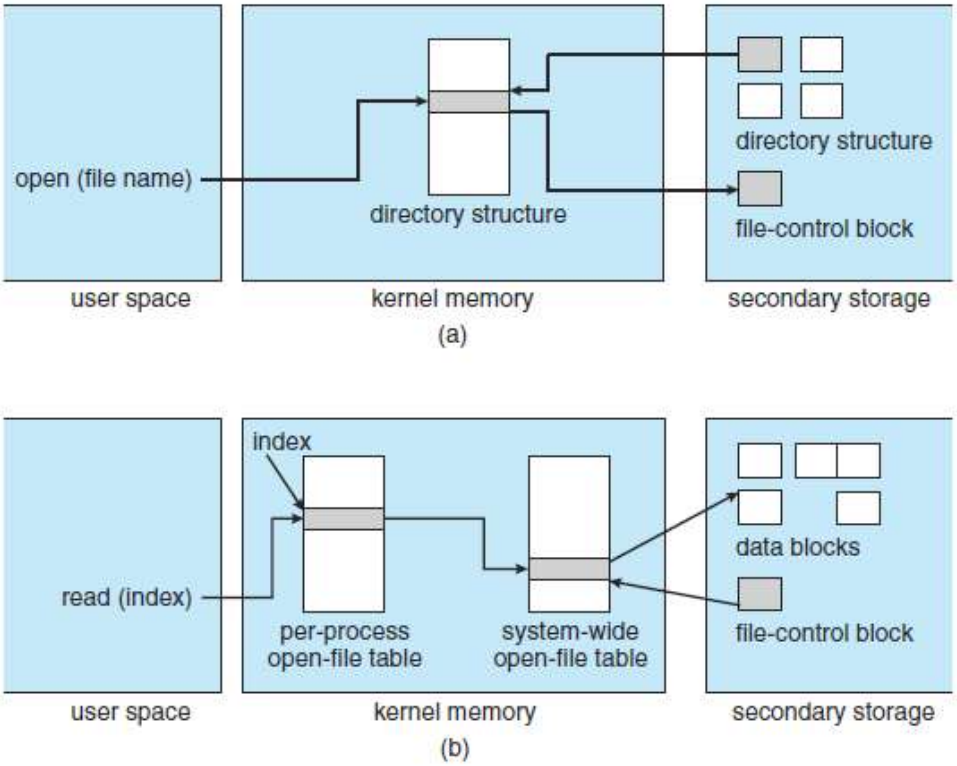


Figure 12.3 In-memory file-system structures. (a) File open. (b) File read.

- Upon close, the per process table entry is deleted, and the system wide entry's open file count is decremented. If all users are done with the file, the system wide entry for that file is deleted as well, and updated metadata is copied back to the disk.
- A partition of a disk can be raw with no file system, or cooked with a file system. A **raw disk** is used when having no file system makes sense, like for swap space or a database.
- Boot info can be stored in a separate partition. Its responsibility, as a collection of blocks that are at the start of the disk, back to back. The **boot loader** is this collection, and knows enough to load the kernel of an OS and to build the file system.
- The **root partition** has the OS kernel and other system file is mounted at boot time. It does validation by reading the disk and verifying that a valid file system exists.
- Most OS use object oriented techniques to implement multiple file systems on the same system. The file system is the first interface that does this, and allows programmers to read, write, open, and close() files. The second layer is the **virtual file system**. This does 2 things:
    - it separates generic file system operations from their implementations (read(), open()). It does this by defining the interface you use in all your code.
    - It provides a mechanism for uniquely representing a file throughout a network, a struct called a vnode. The vnode contains a numerical designator for a network wide unique file.
- Basically, it takes the generic stuff apps use in their library code to edit files and adapt those calls to the required file system operations for the different file system types.

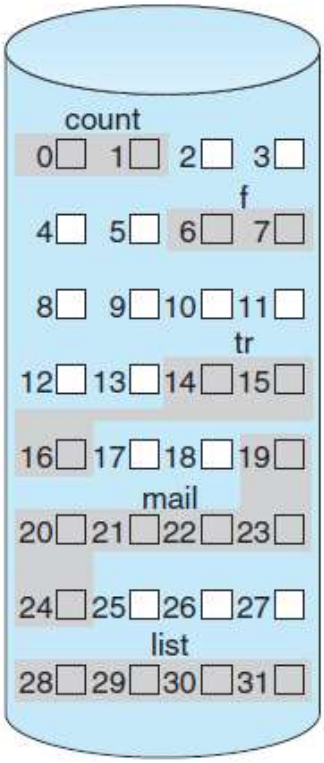
12.3 Directory Implementation

- A **linear list** allows you to have a list of file names pointing to their logical blocks. To create, we search the entire directory for an identical filename, and if it isn't there, we add a new file to the end of a list. To delete, we do a linear search on the list.
- A **hash table** takes a hash value from the file name and returns a pointer to the file name in a linear list which still stores directory entries. This kills the linear search.

12.4 Allocation Methods

- We need a good way to allocate physical blocks (and thus space on the disk) to the logical blocks that hold files. There are three major ways to allocate disk space.
- **Contiguous allocation** requires that each file occupy a set of contiguous blocks on the disk. This means all blocks for a file are stored directly next to each other on the same of a limited number of tracks. This has some problems. Finding space for a new file is hard, as the blocks have to be

kept next to each other.



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

-

Figure 12.5 Contiguous allocation of disk space.

- The contiguous allocation problem can be seen as an application of the **dynamic storage allocation problem** from chapter 8. In contiguous allocation, we usually use first fit or best fit to allocate blocks for files. However, this method suffers from external fragmentation. As files are placed and removed, the free disk space is broken into small chunks. To fix this, we copy the entire file system to another disk, and free the original disk space completely. After this, we copy all of the filled blocks from the copy back to back, and create a large contiguous block of free space, **compacting it all into one space**. More problems exist still. If we do not originally allocate enough space, we cannot fit the file if it grows bigger. File systems will use an **extent**, which is a small amount of contiguous space added onto an already allocated block.
- **Linked allocation** solves these problems. Each file is a linked list of disk blocks, which are scattered across the disk, with a pointer to the first and last blocks of the file. Each block contains a pointer to the next block on disk:

-

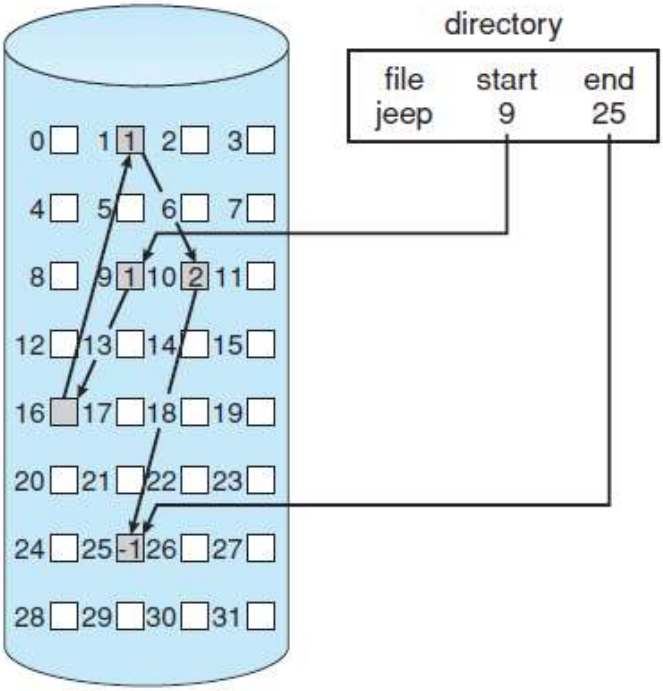


Figure 12.6 Linked allocation of disk space.

- An empty file is one with a null start pointer. However, it is only efficient for sequentially accessing a file. If we want to get to the middle of a file, we have to follow half the block pointers in the file. The fix for this is to use **clusters**, which are collections of blocks (four or so). We only point from the end of one cluster to the next cluster, wasting less space on pointers. This also allows for less time wasted on following pointers and jumping the disk head around. We still kinda suck at direct access though.
- **Indexed allocation** solves the problem of having to follow the pointers by moving all pointers into the **index block**. This makes the directory only index the file name to the block the index block is stored in. We can then search the index block for the middle of the file:

-

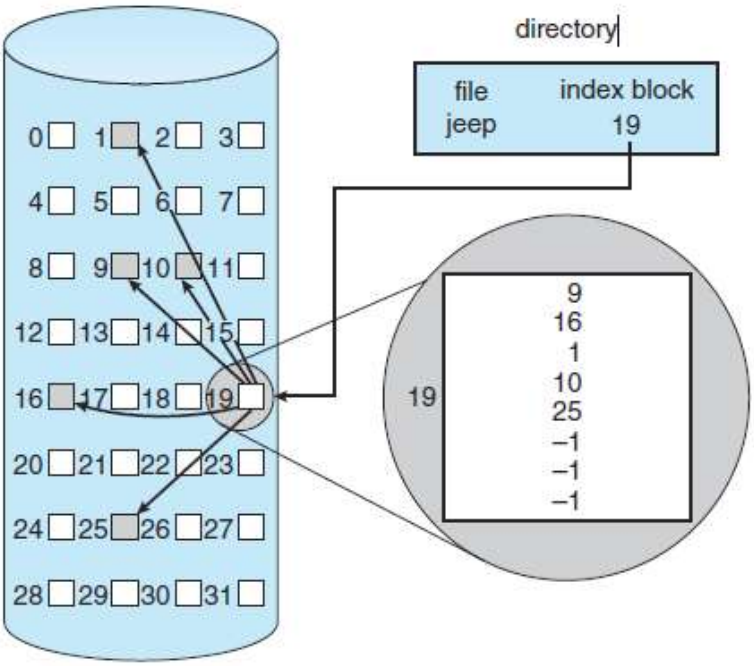


Figure 12.8 Indexed allocation of disk space.

- We waste more disk space with pointers in this scheme, as if we have file that need only one or two blocks, we waste more space with the index block than we would with a pointer on every block. We also need to make an index block large enough for the amount of blocks in a file. We can solve this by linking index blocks together, using a multilevel scheme for index blocks, or other schemes.
- Most OS use a combination of the three schemes to organize their files.

12.5 Free-Space Management

- We need to reuse space from deleted files for new files. To keep track of free space, the **free space list** is maintained by the OS. This list is a list of all the free blocks on the system. To create a file, we search the list for the needed amount of free space and allocate it to a file. When deleting, we find the file in the list, and allow that space to be allocated later.
- A **bit vector/map** is a string of bits, which represent one block on the disk. If it is 0, it is free, if it is 1, it is allocated.
- The **linked list** approach links together all free disk block locations, and saves a pointer to the first free space that can be followed to find the rest of the free space.
- **Grouping** stores the addresses of the first large group of n free blocks in the first free block. The last block in this group stores a pointer to the next large group.
- **Counting** keeps a copy of the first free block of every group, and saves the number of free blocks after the first one.

I/O SYSTEMS (NEED TO DO)

13.1 Overview

- **Device drivers** present a uniform device access interface to the I/O subsystem. They act like a system call interface to the OS.

13.2 I/O Hardware

- Computers have to operate a bunch of devices. A device communicates with the system via a connection point called a **port**. If devices share a common set of wires, the connection between the two is called a **bus**. A bus also has a protocol defining how a set of messages can be sent on the wires. These messages are conveyed by patterns of electrical voltages applied to the wires. A **daisy chain** is when a bunch of devices connect to a port through connections with each other.

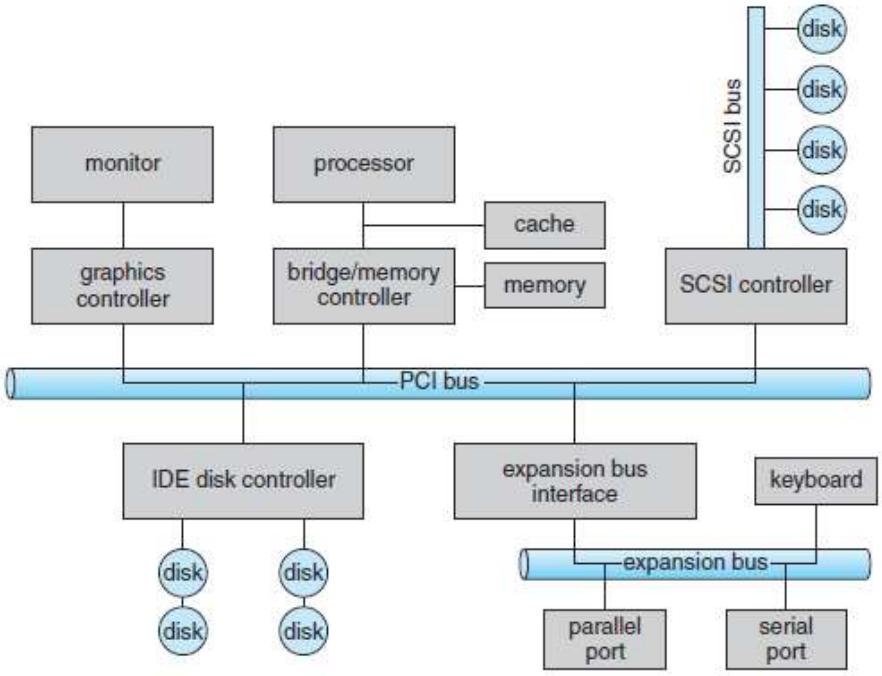
- A **PCI bus** connects the processor/memory subsystem to fast devices, the **expansion bus** connects the keyboard, USB and serial ports.
- The book will try to tell you about SCSI. This is a Scuzzy port. No one uses these. Fuck the scuzzy port. We use SATA cables like grown men. Also, we connect cards with **PCI Express (PCIe)** busses.
- The **controller** is a collection of electronics that can operate a port, bus, or a device. More complex bus protocols demand more complex controllers, which are usually implemented as separate circuit boards (host adapters) that are plugged into the PC. This board usually has a processor and memory as well.
- A disk controller is a good example of a host adapter. It is a board on the hard drive, and controls the SATA protocol over the bus. The main CPU can tell a controller what to do by writing to the onboard registers in the controller. An alternative to this is **memory mapped I/O**, which makes it seem like the controller registers are part of the address space of the processor, meaning we can use memory operations to write to them.
- 

Figure 13.1 A typical PC bus structure.

  - An I/O port typically has four registers, called the:
    - **Status register**, which contains bits that the host can read. These bits indicate states of the port, like when errors occur or a command is being run.
    - **Control register**, which can be edited by the host to start a command or change the mode of a device. Think of this to set modes of the port and the device itself by modifying how the protocol works.
    - **Data in register**, which is read by the host to get input from devices.
    - **Data out register**, which is written to by the host to send info to devices.
  - These registers are usually 1-4 bytes. They can have FIFO queues to hold more info if needed.
  - The actual protocol for communication between the host and controller is really hard. We water it down into the basics with handshaking. The first idea we visit is **polling**. Polling can also be called busy waiting, and it simplifies down to the host repeatedly checking a port register for permission to do work. This can be implemented efficiently, but wastes host cycles waiting.
  - A better idea is to use an **interrupt**, which follows the idea that the device should notify the host when it is ready.

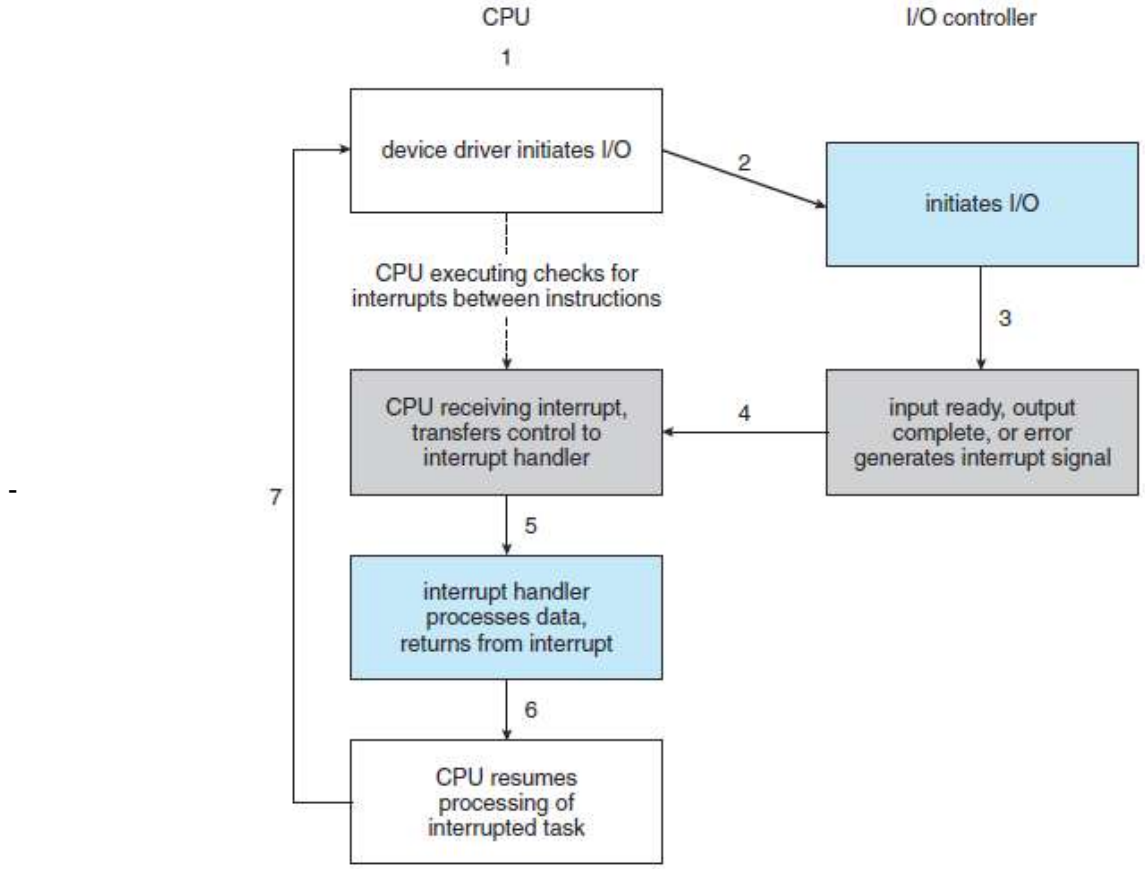


Figure 13.3 Interrupt-driven I/O cycle.

- A CPU has a wire called the **interrupt request line** that the CPU looks at after every instruction. If it detects one, it jumps into an interrupt handler routine after saving its current state. This routine is saved at a constant place in memory. This routine does the right thing based on interrupt type. After this, it goes back to the state that was saved before it did interrupt work. We say a device controller raises an interrupt, and the CPU catches the interrupt and dispatches it to the handler. The handler clears the interrupt by servicing the device. A good example of an interrupt is a single keystroke. Think about how many interrupts have to be handled in a modern PC.
- An **unmaskable** interrupt request line is for interrupts that HAVE to be handled like corrupt memory errors, while the **maskable** interrupt line is there for interrupts that can be ignored by the CPU so critical operations are not stopped. When the interrupt occurs, it passes an **address** to the CPU, which is used to select the interrupt handler routine to run. This address is stored in a table called the **interrupt vector**. This vector can have all the routines needed, or point to other tables of more specific routines. Interrupts can also have priority levels.
- Interrupts are used for a ton of things, like keyboards, disk drives, and even page faults. System calls use interrupts too, called **software interrupts (traps)** that make the interrupt hardware save the state of the code calling the trap and having the kernel perform the system call. These are kind of low priority.
- **Direct Memory Access** allows a device controller to write directly to memory. It does this by asking the CPU for access to the memory bus via an interrupt, and then uses a **DMA controller** as a middleman between the controller and memory. Handshaking between the device controller and DMA controller is done with a DMA request/acknowledge. Then, data from the device goes from the device controller to the DMA controller, and from the DMA controller to the memory over the memory bus. Once the transfer of data is done, an interrupt is raised so that the CPU knows the transfer is done.

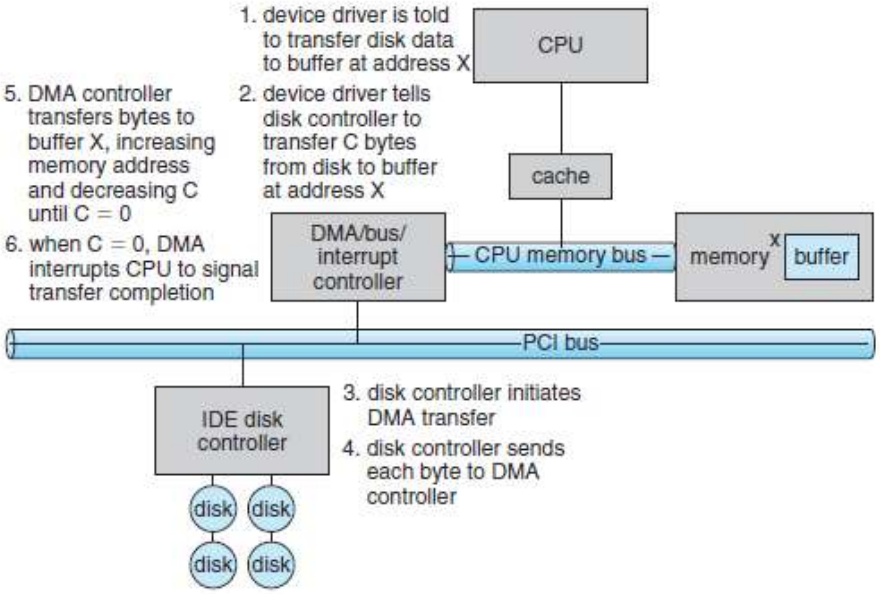


Figure 13.5 Steps in a DMA transfer.

13.3 Application I/O Interface

- This section covers interfaces for the OS to handle I/O devices. Each general kind of I/O device is accessed through a standard set of functions called an **interface**. These are interfaces are usually encapsulated in kernel level modules called device drivers.

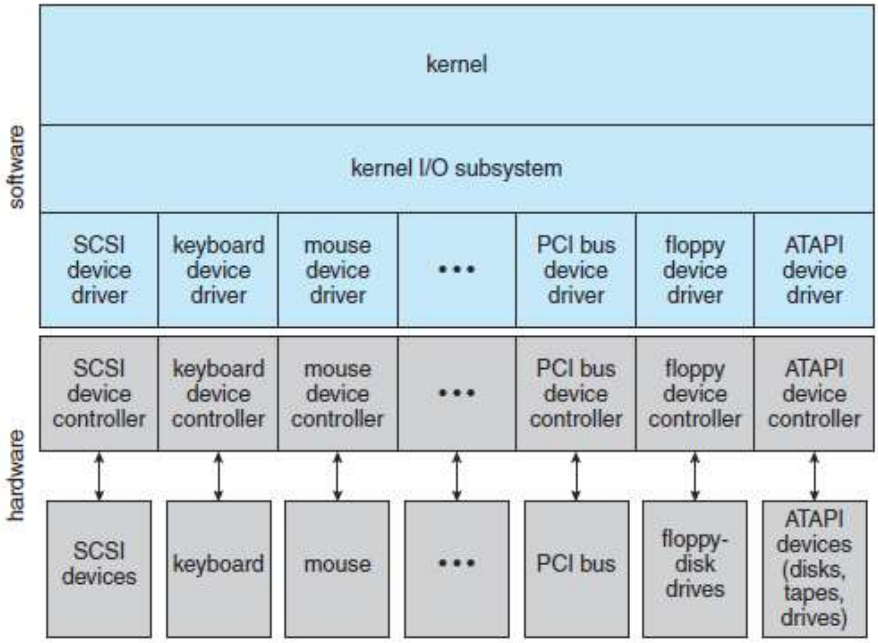


Figure 13.6 A kernel I/O structure.

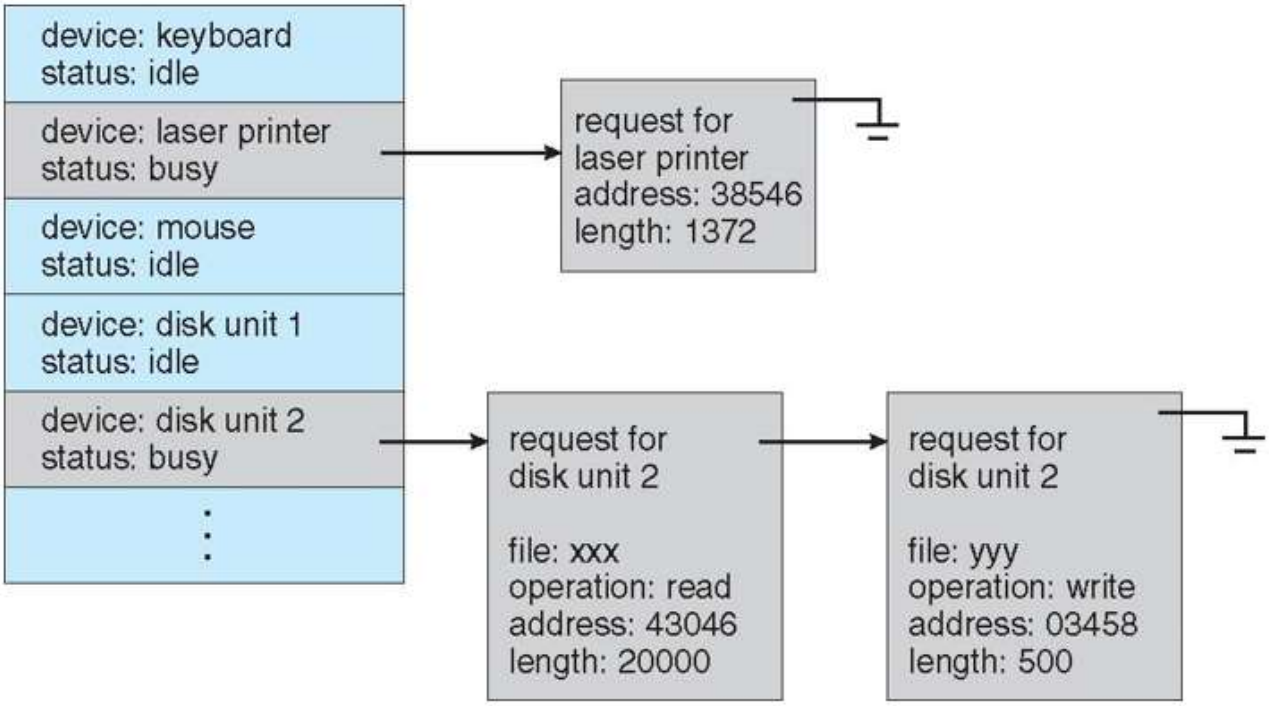
- The device driver is its own layer of the kernel. Hardware makers will either make their device controllers compatible with existing device drivers, or write their own for the OS that will use the device. This allows computer users to use new devices out of the box with little configuration.
- Since each OS has its own ways to implement a device driver interface, code has to be written for each OS to support a driver on each OS. Since I/O devices can vary in many characteristics, a driver is almost always needed.

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

-

- Many OS have an escape that transparently passes arbitrary commands from an application to a device driver. An example of this is the ioctl() command in UNIX.
- The **block device interface** captures all needed aspects for disk drives and other block oriented devices. This is what handles read() and write() commands. We can use this for file system implementation as well. A **character stream interface** is what takes care of commands like puts or gets(). It takes input from a keyboard or even a file.
- A network interface used in most OS are called **sockets**. You know this already, you took 3461.
- Clocks and timers provide the current time, the elapsed time, and a timer. A **programmable interval timer** is used for timing and periodic interrupt cycles. You can use ioctl() for this as well.
- **Blocking** is the act of suspending a process until I/O is done. It is easy to use and figure out, but isn't enough for some systems. **Nonblocking** has the I/O call return as much info as there is available at the time. This is done by running a separate thread that handles the I/O. We can use software to grab data as it comes in. **Asynchronous** I/O has a process run while I/O executes, which is hard and the IO subsystem will signal the process when it is done in full.
- **Vectored I/O** allows one system call to perform multiple I/O operations. Readve() can do this in Unix. This decreases context switch overhead and provides atomicity.
- The kernel I/O subsystem allows scheduling of the device drivers, and **buffering**, which stores data in memory when transferring between two devices. This copes with different data speeds in devices, the size of the bus between devices, and maintains copy semantics. **Double buffering** creates two copies of the data. The kernel also **uses caching** for a faster device holding a copy of data. A cache can only hold a copy of data from somewhere else. **A spool** is a buffer that holds output for a device that can only serve one job at a time. The subsystem keeps track of the devices through the **device status table**, seen here:

-



- The subsystem also handles errors. It will return error bits to system calls, meaning either success or failure.

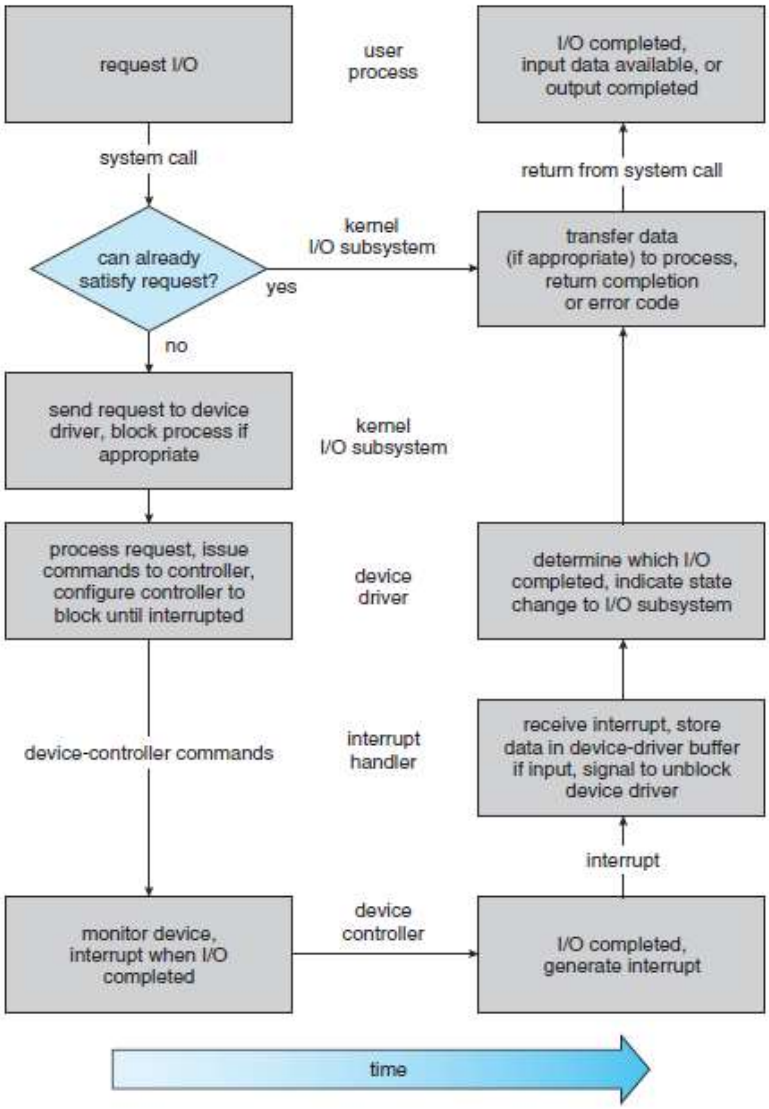


Figure 13.13 The life cycle of an I/O request.

SECURITY

15.1 The Security Problem

- A system is **secure** if its resources are used and accessed as intended under all circumstances. Total security can never be achieved, but we can make breaches rare.
- A threat is an area weak in security, while an attack is an attempt to break security.
- There are multiple forms of security violations:
  - Breach of confidentiality, the unauth'ed reading of data. Grabbing credit cards numbers or passwords.
  - Breach of integrity – unauth'ed modding of data. Think changing source code to print out passwords.
  - Breach of availability – Unauthorized destroying of data. Think putting a big black cock on the White House's homepage.
  - Theft of service – Unauthorized use of resources, like a keylogger.
  - Denial of service – Prevention of legitimate use of a system.
- **Masquerading** is when someone pretends to be someone else to break authentication, and gain access they normally do not have. A **replay attack** consists of a malicious repeat of a valid data transmission, with that data usually modified to escalate privileges or something else.
- A **man in the middle attack** has a bad boy sit in the data flow of a communication, and act as a sender to a receiver on both ends. They masquerade as both of the legitimate parties. A **session**

hijacking usually precedes these, as this is when a communications session is interrupted.

- Security measures must be taken at 4 levels to get security to near totality:
  - Physical – The sites containing must be secured against armed or other entry by intruders. Machines and terminals must be secured.
  - Human – Authorization must be done carefully to make sure appropriate users can get into the system. Social engineering can be done to trick humans into giving other people their access. Tactics like this involve phishing or dumpster diving, which is when you look through physical info for auth info.
  - Operating system – The OS must protect itself from breaches, like a runaway process causing a DoS. A buffer overflow could allow the launching of an unauthorized process, and many other things.
  - Network – Intercepting data on the internet, or interrupting communications.

- Security at all these levels must be maintained to keep OS security ensured. Improving security is like a cat and mouse game, with iterations from both defenders and attackers.

15.2 Program Threats

- A common goal of hackers is to write a program that creates a breach. A back door, for instance, allows a way into a system without authorization easily after the exploit that allowed it to be created in the first place is blocked.
- A **trojan horse** is a code segment that misuses its environment. It can be used to grab data, delete data, and mess your day up.
- **Spyware** is a variation of the Trojan, which plays ads for a user involuntarily, or captures information from the users system and sends it to a central site. This latter thing is a category of attack called **covert channels**, with bad communication doing evil things.
- A **trap door** is something a programmer leaves in software only they are capable of using.
- A **logic bomb** is a program that initiates a security incident under certain circumstances. Like when a programmer is fired, the entire database is deleted, along with all backups.
- A buffer overflow attack is the most common way an attacker outside the system gains unauthorized access to the system. To do this, a series of steps must be followed:
  - Overflow an input field, command line argument, or input buffer until it writes into the stack. This is done by providing more info than the program was (poorly) programmed to expect.
  - Overwrite the current return address on the stack with the address of exploit code.
  - Write a simple set of code for the next space in the stack that includes malicious commands for the hacker.
- A stack frame contains things used in a single function within an executing program. You can find local variables and parameters here, as well as the return address of the calling function, and the frame pointer, which is the address of the beginning of the stack frame.
- A **virus** is a fragment of code embedded in a legit program. They self replicate and infect other programs, and cause damage. A virus dropper inserts the virus into the system, with the dropper usually being a trojan. It can do many things:
  - File viruses infect a system by appending itself to a file, changes the start of the program to execute its code, and then resumes normal operations for that file.
  - A boot virus infects the boot sector of the system, loading every time we boot the OS.
  - Macro viruses are viruses that are programmed in a high level programming language.
  - Source code viruses look for source code, modify it, and uses the code when it is built to spread itself.
  - A polymorphic virus changes each time it is installed to avoid detection. A virus signature is what is changed by this, and this is a pattern that can be used to ID a virus, like a series of bytes.
  - An encrypted virus decrypts itself, runs and reencrypts.
  - A stealth virus modifies parts of the system designed to detect it .
  - A tunneling virus attempts to bypass detection from an antivirus by installing itself in the interrupt handler, or device drivers.
  - A multipartite virus infects many different parts of a system, like boot and files.
  - An armored virus is coded to make it hard for antivirus guys to unravel and figure out.
- A **monoculture** is one computer culture in which systems run the same hardware, OS and

software. This is basically Windows.

15.3 System and Network Threats

- A **secure by default** OS disables every insecure feature at install and requires people to turn them on if they want them. An **attack surface** is the set of ways in which an attacker can try to break in.
- A **worm** is a process that uses the spawn mechanism to duplicate itself. This is usually done to use up resources or lock out all other processes. The Morris worm used a grappling hook program, which would be run on the attacked computer, and the main worm, which the hook program downloaded and then executed on the attacked computer. The hook originally got onto the computer using a remote shell exploit.
- **Port scanning** is a method of scanning the range of ports on a machine for exploitable programs. Port scans are detectable, so they are usually run off of zombie systems, those systems previously hacked. If an attacker uses the zombie it has to scan a targets ports, its harder to trace.

15.4 Cryptography as a Security Tool

- **Cryptography** is a tool used to limit the number of computers in a network that can process specific messages. **Keys** are the tools that usually designate which computers can process those messages.
- An encryption algorithm has to determine a message into a ciphertext that can only be computed when the receiver has the correct key to compute the message from the ciphertext.
- Symmetric encryption uses the same key to en/decrypt. This means the key K must be protected at all times. DES and AES are symmetric ciphers.
- Asymmetric encryption uses different keys for en/decrypting. Someone who wants encrypted content passes their public key out to the world. This key is used by senders to encrypt messages. Only the private key corresponding to the public key can unlock those messages.
- RSA used a prime factorization algorithm to generate keys.
- **Authentication** is the constraining of the set of potential senders of a message. We use it to verify a message or program hasn't been manipulated. IN a **message authentication code**, a hash function (usually crypto) is run against a file, and if the message output matches the one from the sender, you know no tampering happened.