CMSC 421 Study Guide

Corey Atkins, Max Poole, Sherwin Fong

Ch.1: Overview

What is an OS?

A program that acts as an intermediary between a user of a computer and the computer hardware.

What is a kernel?

The one program that is running at all times on the computer.

Computer system organization and architecture

Organization has CPUs and device controllers connected through buses. Architecture is made up of multiprocessors (Asymmetric or Symmetric). They increased throughput, increased reliability, and better economically.

OS structure

- multiprogramming and timesharing

Multiprogramming keeps CPU busy at all times, is used for efficiency. Timesharing (multitasking) allows a CPU to switch jobs so frequently that a user can interact with each job while it's running.

OS operations

Operations such as CPUs executing concurrently, moving data f/t main memory t/f local buffer, and device controllers informing CPU it's done its operation by causing an interrupt.

Process management

A process(active) is program(passive) in execution. Has resources such as CPU, memory, and I/O files. Single-threaded processes executes instructions, one at a time. Multi-threaded processes execute with concurrency. Manages all processes.

Memory management

Manages all data in memory before and after processes, all instructions in memory to execute, determination of optimizing CPU, etc.

Storage management

Manages files and directories.

Protection and security

Protection for controlling access of processors or user resources. Security defends against internal and external attacks.

Some basic data structs used in the kernel
Singly, doubly, and circular linked lists, BST, Hash Maps

Ch.2: Operating System Structures
OS services (10)
- UI, program exec., I/O, file system, communications, etc.

UI is either command line or GUI. Program exec must load program in memory, run program, and end execution. I/O can be a file or device. File sys must manipulate files. Comms processes exchange info. Error detection must be aware of errors. Resource Allocation give resource to multiple concurrent users and jobs. Accounting keeps track of the number of users and available resources. Protection ensures all access to system resources are secured. Security defends against outside attacks.

User mode and kernel mode
User Mode (Non-zero). Kernel Mode (0). Program has to notify OS when done. Timer to prevent infinite loop. An interrupt is set up for a specific period, OS will decrement counter, and when it hits zero, it will generate an interrupt.

User's interface to the OS: system calls
System call change mode to kernel, return from call resets mode to user.

Types of syscalls (6)
Process Control (managing processes)
File Management
Device Management
Information Management
Communication
Protection

Interrupt-driven nature of OSs
OS is driven by interrupts because interrupts transfer control to the interrupt service through an interrupt vector table, which has addresses to all service routines. A trap or exception is a software interrupt caused by a user request or an error, respectively.

System programs (9)

Status Info
Helper programs that make system calls.
Communications
File manipulation
File modification
Programming language support
Program loading and exe
Application programs
Background services

OS design and implementation
Best OS is not achievable. User Goals: convenient use, easy to learn, reliable, safe, fast. System Goals: easy to design, implement and maintain, flexible, reliable, error-free and efficient. Policy: What will be done? Mechanism: How to do it? More mechanism and less policy (policies can change later). Linux kernel is written in C. Kernel is written in assembly because it is fast.

OS structure--multiple alternatives
  - simple (MS-DOS), UNIX, fully-layered, microkernel, etc.

Simple (MS-DOS) was written to provide the most functionality in the least space. Not divided in modules and levels of functionality are not separated.

Unix has two parts, system programs and the kernel (provides core services) (everything below system call interface and above physical hardware).

Layered Approach, OS is divided into layers, to access a layer that is a couple of positions lower than the current process, it first needs to talk to the layer right below itself.

Microkernel (based on Mach OS) communication takes place between user modules using message passing in the kernel, but downside is every program must deal with un/packing messages and other programs. Modules is most modern, each component is separate and can be loaded by kernel when needed, similar to layers but more flexible.

OS debugging
Log files contain error info to find issue. Core dumps (App Failure) capture memory of the process. Crash dump (OS Failure) capture kernel memory of the process. Performance tuning is used to optimize system performance; such as profiling, a periodic sampling of instruction pointer to find system trends.

System boot process, from BIOS to fully running system
When powering on system, execution starts at a fixed memory location. The bootstrap loader (stored in ROM or EEPROM) locates the kernel and loads it into memory. After the kernel is loaded, the system is then running. Linux Boot Sequence: BIOS (Basic IO Sys) -> MBR (Master Boot Record) -> GRUB (Grand Unified Bootloader) -> Kernel -> Init -> Runlevel.

<u>Ch.3: Processes</u>
Concept of a "process"
A program in execution, process execution must progress in sequential fashion. A program becomes a process when the executable file is loaded into memory.

Process structure in memory
Stack: temp data
Heap: dynamic memory during run time
Data: global vars
Text: program code

Process states and transitions
New , Running, Waiting, Ready, Terminated
New -> Ready = Admitted
Ready -> Running = Scheduler Dispatch
Running -> Terminated = Exit
Running -> Ready = Interrupt
Running -> Waiting = Event wait
Waiting -> Ready = Event Completion

Process control block (PCB) (8)
Process Number: All interprocess comms and actions are done via process number.
Process State
Program Counter: Location of next instruction to execute
CPU Registers: contents of all registers
CPU Scheduling info: priorities, scheduling queue pointers
Memory Mgmt info: allocated memory for process
Accounting info: CPU used, clock time, etc.
I/O Status info: I/O devices allocated to process

Process scheduling
We want to maximize the CPU usage. Selects available processes for next execution on CPU

Queues
Job Queue: all processes in system
Ready Queue: set processes in memory, waiting to be executed
Device Queue: set processes waiting for I/O device

short-term and long-term schedulers
Short term: selects process that should be executed next (fast)
Long term: selects process that should be brought into ready queue (slow)

Context switch
Saves the state of the old process and load the state of the new process when switching processes in CPU. Represented by PCB.

Operations on procs
Process creation and termination.

UNIX fork()/exec(), parent/child proc. Relationships
Parent process creates child.
Parent can share either all, some, or none resources with children.
Parent can either execute with children or wait until children terminate.
Child's address space is a duplicate of parent's. Also child can have a program loaded into it.
fork(): creates new process
exec(): used after fork() to replace process memory space with new program (used by child and parent must wait)

Process needs to ask OS to delete it.
Process is terminated either by exit() or abort().
Parent must wait() for child to exit(). Resources get deallocated.
If parent don't wait(), then process is a zombie.
If parent is terminated, processes are orphans.

Interprocess communications (IPC)
- implementations: shared mem, message passing

Shared Memory: There is a shared space in which one process read to and another reads from.
Message Passing by Kernel: One process puts message in the queue and the other pulls them out.

Producer/consumer problem
- Bounded buffer solution

Create a struct of the item. Then create an array of the item and set size to the buffer - 1.

Direct vs. indirect IPC
Direct: Processes need to be named explicitly. Links automatically established. Link is between a pair of processes, usually bi-directional.

Indirect: Messages are sent and received through ports (mailbox). Processes can communicate if they share a port. Links can be with many processes, and pairs can have several links, either unidirectional or bidirectional. Must be able to create, send, receive, and destroy mailboxes.

Indirect Solution: Link can have at most 2 processes, with executing one process at a time. System can select some receiver and sender will be notified who it was.

Synchronous vs. asynchronous communications
Synchronous (Blocking): process is put to sleep until someone has received the message.
Asynchronous (Non-blocking): does not have to wait. Can read message without waiting.
Computers are typically asynchronous writing and synchronous reading.
If both send and receive are blocked, then it is rendezvous.

Sockets, remote procedure calls (RPCs)
- especially UNIX pipes

Socket: an endpoint for communication. Concatenated with an ip address and port number. Types of sockets are connection oriented (TCP), connectionless (UDP), and MulticastSocket class.

Remote Procedure Calls: abstract procedure calls between processes on a network. Stubs is a client side proxy for actual procedure on server. Client side locate server and parameters. Server side receives message, unpack parameters and performs procedure on server.

External Data Representation (XDL) formats for different architectures, like Big Endian and little endian.

Pipes: allows for interprocess communication, only unidirectional. Producer write to one end and consumer reads from the other end. Has parent-child relationship.
The system call pipe() takes an address of an array of two ints (fd). The ints are indices to the file descriptors table. System call creates a pipe and

its two file descriptors, one for reading and the other for writing. Note: CANNOT read while writing and write while reading. Fork() shares all open files, then parent and children can communicate without knowing pids.
Parent: fd[1]
Child: fd[0]

<u>Ch.4: Threads</u>
What is a thread?
A execution of code or a lightweight process. Has its own program counter, system register and stack. Shares data and files with other threads.

Motivation for using threads
Creating threads is more lightweight than creating separate processes, they can also run on applications, do multiple tasks, simplify code and increase efficiency.

Multicore programming
Issues:
- **Dividing activities**
- **Balance**
- **Data Splitting**
- **Data Dependency**
- **Testing and debugging**

Parallelism: run more than one task simultaneously
Concurrency: supports more than one task making progress

Data Parallelism: distribute subsets of same data across different cores but with same operations on each
Task Parallelism: distribute threads across cores, each thread is its own unique operation

CPUs have both cores and hardware threads (typical some number of threads per core)

Memory structure of multithreading
Single-threaded process: has registers, a stack, code, data, and files.
Multi-threaded process: each thread has own registers and stack, but shares code, data and files.

Multithreading models

Many-to-One: Multiple user threads to one single kernel thread. One thread blocking causes the whole thing to block. Cannot have multiple threads in parallel since only one thread at a time. Not used often.

One-to-one: One kernel thread per user thread. Can run in many parallel but lots of overhead. Most common.

Many-to-Many: **Many user threads can be mapped to many kernel threads. Not used often.**

Thread libraries, e.g. POSIX Pthreads
Thread libraries allow a user to create and manage threads
POSIX is a standard for thread creation and synchronization. Pthreads is a thread library, it is only an abstraction not an implementation.

Implicit threading
Thread creation done by compilers and run-time libraries rather than programmers. Methods include: Thread Pools, OipenMP, Grand Central Dispatch

Thread Pools: Create a bunch of threads in a pool where they wait for work. Can be faster to service a request with an existing thread than to create a new one.

OpenMP: Set of compiler directives and an API for C/C++ and Fortran. Supports parallel programming in shared-memory environments by identifying *parallel regions*.

Grand Central Dispatch: Apple Technology. Extensions to C/C++, API, and runtime library. Allows identification of parallel sections and manages most of the details of threading. Blocks are placed in a dispatch queue that is either a) serial: Removed in FIFO (first-in, first-out) order, or b) Concurrent: Removed in FIFO order but can remove more than one at a time

Threading issues

- **Semantics of fork() and exec() system calls**
    - **Fork(): Does fork duplicate only the calling thread or *all* threads (some UNIX-based OS's have two different versions)**
    - **Exec(): Usually works as normal - replace the running process including all threads**

- **Signal Handling: processes signals. Generated by an event, delivered to a process, or handled by default or user defined signal.**

- **Thread Cancellation of Target Thread: Asynchronous terminates immediately. Deferred periodically checks if need to be cancelled, until it reaches cancellation point.**

- **Thread local storage (TLS): allows each thread to have its own copy of data. Useful when do not have control over creation process. Different from local variables, very similar to static data.**

- **Scheduler Activations: uses a lightweight process (LWP) as a intermediate data structure between user and kernel threads. Uses upcalls to provide communication between kernel and upcall handler in thread library.**

OS e.g.s
Windows XP threads: One-to-one mapping, kernel level. ETHREAD, KTHREAD, TEB
Linux threads: Uses clone() to create threads. CLONE_FS, CLONE_VM, CLONE_SIGHAND, CLONE_FILES.

<u>Ch.5: Process Synch</u>
Race condition
- Should be familiar with extended Bounded Buffer problem, trying to use every buffer element

A situation where two conditions are "racing" to complete the same task. One process wants to write and read before the other process.

Critical-section problem
Entry Section: enters critical section.
Critical section: a segment of code where no other process can run its code.
Exit Section: exits critical section.
Remainder Section: code waiting to enter critical section.

- Necessary properties of any complete solution to the critical-section prob.
Mutual Exclusion: if one process is in critical section, no other process can enter their critical section.

Progress: if no process is in critical section and another process wish to enter, then we cannot stop it from running forever.

Bounded Waiting: a bound to limit the number of times a process can enter its critical section; after a process made a request to enter its critical section and before request is granted.

Preemptive: in kernel mode, a process can be stopped.
Non-preemptive: runs until exits kernel mode.

Peterson's soln
Good solution to the problem, but isn't actually used in real life.

```
do {
   flag[i] = true;
   turn = j;

   while (flag[j] && turn == j);
   /* do the critical section */
   flag[i] = false;
   /* remainder section */
} while(true);
```

Assumes load and store are atomic.
Variable turn determines which process can enter the critical section.
The flag determines if a process is ready to enter the critical section.

Synchronization hardware
 - Atomic operations, and why they're important
Atomic operations are important for thread safe applications. If something is atomic, it happens entirely or not at all. With threads this is important because we want our data to be atomic, and have no unexpected behavior.

- pseudocode version of test-and-set, compare-and-swap atomic ops
```
boolean TestAndSet(int* var) {

  int temp = *var;
  *var = 1;          // Sets to True
  return temp;
}

int CompareAndSwap(int *actual, int expected, int newVal) {

        int temp = *actual;
        if (*actual == expected) {
                *actual = newVal;
        }

        return temp;

}
```

Mutex locks using spinlocks

- acquire() and release()

Acquire() spins a while loop that does nothing until the lock is available (it spins for a lock therefore spinlock). Release() gives up the lock, so acquire() will spin until some other thread calls release(). The issue with this is that we are wasting CPU cycles spinning around doing nothing.

Semaphores

Similar to a mutex but allows for more processes to access data at once. Lock decrements when a process acquires the lock. Increments when the process releases the lock. If value of the lock is 0, then no other process can run.

- wait()/signal() (or P()/V())

Wait() will wait until the value of the lock is not zero, then it will decrement the lock by 1. Signal() will increment the lock by 1.

- both counting semaphores, and mutexes through binary semaphores

Binary semaphore: a semaphore where the lock can only be 1 or 0, is a mutex lock.
Counting semaphore: a semaphore where the lock can be any range.

- semaphore implementation with no busy waits: using block() and wakeup()

Each semaphore has a waiting queue.
Block: places a process on the waiting queue
Wakeup: will remove a process from that waiting queue, and place it in ready queue.

```
/*********************************************************
******
typedef struct{

 int value;
 struct process *list;

} semaphore;

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);
         }
}
*********************************************************
*****/
```

Deadlock and starvation

Deadlock: when two or more processes are indefinitely waiting for each other to cause some events.
Starvation: a process that may never be removed from semaphore queue, if suspended.

- should be able to give examples

In a real world example we could imagine two very polite people in line. The person in front will not get his food until the person behind him gets her food. The person behind him will not get her food until the person in front of her gets his food. Because both people are waiting for each other, no one will get their food.

Replace those people with processes and replace the food with some resource in memory and you get a good illustration of deadlocks.

- some (possibly unrealistic) solutions to deadlock

Classic problems of synchronization:
n buffers, each hold 1 item
Semaphore mutex init to 1
Semaphore full init to 0
Semaphore empty init to n

- Bounded-buffer problem, solution w/semaphores

Producer and consumer both share a fixed size buffer.

Problem: Producer does not add data to a bounded buffer if it is full. Consumer does not try to remove data from an empty buffer.

Solution: Producer goes to sleep if the buffer is full. If the consumer removes data from the buffer, it wakes up the producer. Similarly, the consumer sleeps if the buffer is empty, and can be woken up by the producer as soon as the producer puts new data into the buffer.

```
/***********************************************************
******
Producer does:
do {

/* produce the next item */

wait(empty);
wait (mutex);

/* now add that item to the buffer after acquiring the lock */

signal(mutex);
signal(full);             // Goes to sleep

} while (true);
***********************************************************
*****/
```

```
/***********************************************************
******
Consumer does:
do {

wait(full);
wait(mutex);

/* remove an item and put it into next_consumed */

signal(mutex);
signal(empty);             // Goes to sleep

/*consume the next item in next_consumed */


} while (true);
***********************************************************
*****/
```

- Readers and writers problem, solution w/semaphores

We have a shared data set. Readers can only read from the data set and writers can both read and write.

Want multiple readers to access data at the same time, while only one writer should be able to access the data.

rw_mutex init to 1
mutex init to 1
read_count init to 0

```
/***********************************************************
******
Writer:
do {
   wait(rw_mutex);

   /* writing is performed */

   signal(rw_mutex);
} while(true);
```

***/**

```
Reader:
do {

   wait(mutex);
   read_count++;

   if (read_count == 1)
       wait(rw_mutex);
   signal(mutex);

    /* reading is performed */
    wait(mutex);
    read_count--;

    if (read_count == 0)
       signal (rw_mutex);
    signal(mutex);
} while(true);
```

***/**

- Dining philosophers problem (not solved with semaphores)

There is shared bowl of rice (shared data). Each philosopher (process) has to their right one chopstick (kinda complicated metaphor honestly but is similar to a lock), so there are an equal number of chopsticks as their are philosophers. To eat the rice, a philosopher must have two chopsticks. To drop a chopstick that the philosopher picked up, she must eat some rice first. A problem occurs if all philosophers grab a chopstick at the same time. Then no philosopher will eat and no philosopher will ever drop their chopstick. We have circular deadlock. The solution uses monitors.

Monitors
- general definition

A high-level abstraction that provides a convenient and effective mechanism for process synchronization. This is an abstract data type where we only have access to internal variables. Only one process can be active at a time.

- high-level (i.e., not very detailed) understanding of condition vars

Can use a block on the variables, each has its own queue. Multiple processes can use our monitor and when they approve a critical section, they go back in queue.

x.wait(): process is suspended until x.signal()
x.signal(): resumes one of the processes invoked by x.wait()

If P invokes x.signal while Q is in x.wait(), P must wait.
Signal and wait: P wait until Q leaves
Signal and continue: Q waits until P leaves

- Using monitors to solve Dining Philosophers

Pickup(): test if you can eat(wake up) or not(block).
Putdown(): called after your state is eating. First state is thinking, then test left and right neighbors.
Test(): if both neighbors are not eating and you are hungry, then state is set to eating, signal yourself, then pickup() can be completed.

Sync e.g.s
Prior to kernel 2.6, disables interrupts to implement critical sections. Later versions are fully preemptive (Linux is reading for block and prepared for interrupts).

Alt. approaches
System Model ensures that operations are a single logical unit of work, entirely or not at all. Transaction is a collection of operations that perform a single logical function.

<u>Ch.6: CPU Sched</u>
Purpose of scheduling
Maximize CPU usage.

Type of CPU Scheduling

- **Preemptive: Can cut a task short, stop a process before it is finished to schedule other processes**
- **Nonpreemptive: Waits until a process ends naturally**

Scheduling criteria: factors we can maximize/minimize

- **CPU utilization - keep CPU as busy as possible (else we just waste cycles waiting)**
- **Throughput: # of completed processes/time unit**
- **Turnaround Time: time is takes to execute process**
- **Waiting Time: Amount of time a process is stuck in the waiting queue**

- **Response Time**

Maximize CPU utilization, Throughput. Minimize Turnaround time, waiting time and response time

Sched algs:

- First Come/First Serve (FCFS): **The waiting queue is just a normal queue. The first process that says it is ready will get started first**
- Shortest-Job-First (SJF): **A priority queue where the priority is how long it would take a job to complete. This is the optimal scheduling but it is unobtainable.**
  - with preemption: becomes "shortest-remaining-time-first"

CPU burst prediction

T_{n+1} = alpha(t_n) + (1 - alpha) * T_n
alpha = some weight, commonly set to 0.5
T = guessed time
t = actual time

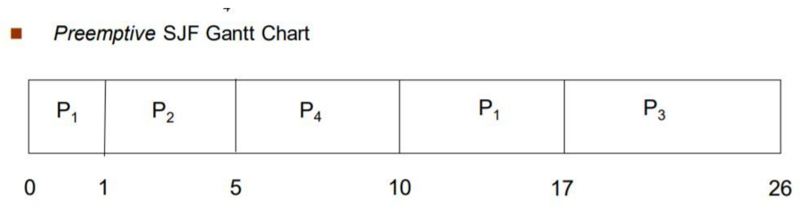
Exponential averaging

T_{n+1} = alpha(t_n) + (1 - alpha) * (alpha * t_{n-1} + $(1 - \text{alpha})^2$ * T_{n-1})
The exponent is n+1.

Be able to draw Gantt charts

Gantt charts show when a process is running. They show the start and stop of the process burst.

Example:



Priority Scheduling

Scheduling using a priority queue. Each process has an integer associated with it, the smallest integer is the highest priority. Note that shortest time remaining is a priority scheduling algorithm where the "shortest time" is the priority. We can also have priority scheduling based on how important a process is. For example we can prioritize processes from the os over a process running a song in itunes.

Round Robin -- implies preemption
Each process gets a time quantum. When a process has used up its time quantum then we go to the next process. If our time quantum is very large (bigger than largest burst time) than we will just have first in first out (FIFO) (meaning that the process will just complete its burst within the bounds of its time quantum). If our time quantum is too small with respect to context switch, then overhead is too high

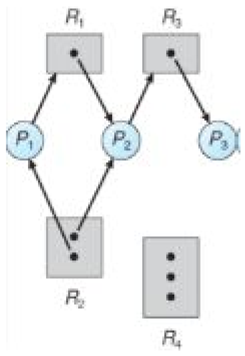
Multi-level Scheduling
We have multiple queues, we can give each queue its own scheduling. Foreground/ Interactive (Round Robin). Background/ Batch (FCFS).

Processor Affinity --just basic concept and motivation
This is the idea that our process has affinity (it prefers) the processor that it is currently running on over other processors. The motivation behind this is that it is expensive to switch processors.

Real-time Scheduling--basic concepts
A process that is able to promise you that it take at most some time.
Real-time processes are reliable, not necessarily fast.
Real-time scheduling must support preemptive, priority scheduling. For hard real-time we must also have the ability to make deadlines.

Scheduling algorithm evaluation
More important ones:
- deterministic modeling
    - **Give each algorithm a defined workload. Compare the results for each.**
- simulation
    - **Program a model of a computer system and run your algorithms on that. Ensures that outside forces will not affect process.**

<u>Ch.7: Deadlock</u>
Resource allocation graphs



P1, P2, P3 are processes
R1, R2, R3, R4 are resource type with a number of instances (dots)
P1 requests instance of R1
P1 is holding an instance of R2

If graph has no cycle:
No cycles = No deadlock

If graph has cycle:
1 instance per resource type = Deadlock
Several instances per resource type = Possible deadlock

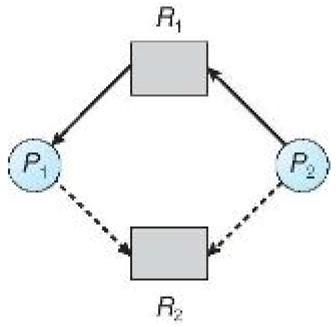
Handling Deadlocks:
· Prevention vs. Avoidance**: ensures that system will NEVER enter a deadlock state.**

· Deadlock Prevention: the four conditions
- **Mutual Exclusion: not required for shared resources. Must hold for non-sharable resources.**
- **Hold and Wait: must guarantee that when request a resource, it does not hold any other resources. Process must request and allocate all resources before execution begins, or request resource when process doesn't have any.**
- **No Preemption: If a process is already holding resources, and request another resource that cannot be immediately allocated, then release all resources. Preempted resources are added to the list of resources that the process is waiting for. Process will restart when it gains both old and new resources.**
- **Circular Wait: impose a total ordering of all resource types. Each process request resources in increasing order.**

· Deadlock Avoidance: **requires a system that has additional a priori information available. Each process but declare maximum number of resources it will need.**

Safe state definition: **if there exists a sequence <P1 … Pn> of all processes in the system such that for each Pi, the resources that Pi can request can be satisfied by current available resources + resources held by all the Pj with j < i.**
Safe state = No deadlocks
Unsafe state = Possible deadlock
Avoidance: ensures system will never enter an unsafe state.

Single-instance algorithm: Resource-allocation-graph algorithm



-Claim edge converts to request edge when process request resource
-Request edge converts to assignment edge when resource is allocated
-Assignment edge reconvert to claim edge when resource is released by process
-Resources must be claimed a priori in the system

Multi-instance algorithm: Banker's algorithm (**HW3 Question 3)**
Each process must have a priori. When process request resource, it must wait. When process gets all resources, it must return them in a finite time.

Need[i,j] = Max[i,j] – Allocation[i,j]

Safety Algorithm
1) **Work = Available**
2) **Find an I such that both:**
    -**Need$_i$ <= Work**
    -**If no I exist, go to step 4**
3) **Work = Work + Allocation$_i$**
    -**Go to step 2**
4) **Finish[i] == true for all i**

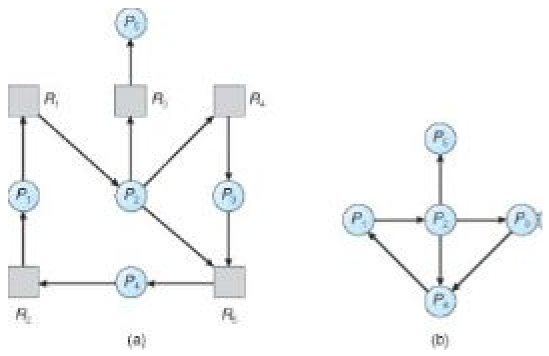
Resource-Request Algorithm
1) **Request$_i$ <= Need$_i$**
    a. **Go to step 2**

2) **Request$_i$ <= Available**
   a. **Got to step 3**

3)
   a. **Available = Available – Request**
   b. **Allocation$_i$ = Allocation$_i$ + Request$_i$**
   c. **Need$_i$ = Need$_i$ - Request$_i$**

· Deadlock Detection: **Allows system to enter deadlock state**

Single-instance: wait-for graph



Part b
Periodically invoke algorithm that searches for a cycle. Cycle = deadlock
Algorithm requires an order of O(m x n^2) operations to detect a deadlock where n is the number of vertices.

Multi-instance: related to Banker's alg.

1) **Work(m) and Finish(n) are vectors**
   -Work = Available
   -For I = 1, 2, … , n, if Allocation$_i$!= 0, then Finish[i] = false, otherwise true

2) **Find an index i:**
   -Finish[i] == false
   -Request$_i$ <= Work
   -If no i exist, go to step 4

3) **Work = Work + Allocation$_i$**
   -Finish[i] = true
   -Go to Step 2

4) **If Finish[i] == false for 1 <= i <= n, then system is in deadlock state and P$_i$ is deadlocked.**

Ch.8: Memory Management
H/W versions:
Protection: base and limit registers

For a logical address space we have base and limit registers. Protection makes sure that the address being referenced for the process is always within [base, base + limit]
Base Registers: holds smallest legal physical memory address
Limit Registers: specifies size of range

Strategies for relocating code
For relocating code we have relocatable addresses. These addresses are defined as an offset away from some base address. For example the relocatable address may be defined as "15 from the beginning of the module." For these to be used we need to have a way to map a relocatable address to a physical memory address. This can be done in three ways
At compile time: When you compile a program assign everything a physical memory address. Generates absolute code, but must be recompiled if the process is moved.
Load: Generates relocatable code at compile time that can be used when the process is loaded into memory.
Execution: Binding from relocatable to physical address is delayed until the actual code is executed.

Logical vs. Physical address spaces
Physical: Address seen by the memory
Logical: Address generated by the CPU (also known as Virtual)
These are the in compile time strategies, but differ in execution strategies.

Memory Management Unit (MMU)
The MMU maps virtual addresses to physical addresses.

Relocation register
The Relocation register is the same thing as the base register

Swapping
Swapping removes the process temporarily from memory and thing brings it back at a later time to continue execution.
The issue with swapping is that context switches can be very expensive and it can be hard to deal with I/O.

Continuous memory allocation: first fit, best fit, worst fit
Contiguous memory allocation puts processes into memory as data blocks that are next to each other. Blocks of memory that do not have a process in them are called "holes."
Three methods of memory allocation

First fit: find the first hole that will fit the process. This is the fastest method
Best-fit: Find the smallest hole that will fit the process. This does the best at storage utilization
Worst fit: Find the largest hole. This tries to leave the largest possible holes left. This aims to reduce external fragmentation

Fragmentation: internal vs. external
External: holes between data blocks
Internal: Space allocated for but not used by processes.
For example say that there is 50 space and a process wants 25. We give the process 40. This means there is 10 lost to external fragmentation and 15 lost to internal fragmentation.

Segmentation
Segmentation supports the user view of a process, by splitting the allocation of the process into allocating for individual components of the process. For example we allocate the local variables and methods in separate data segments.

H/W: segment table structure
In Segmentation Architecture we have a logical address which is expressed as a segment number and an offset. A segment table maps this into a base and limit register just like in contiguous allocation.

Paging
  Basic concept, comparison to contiguous relocation and segmentation
Unlike in contiguous relocation, in paging we can have non-contiguous allocation. This means that we split memory into equally sized physical memory frames and we split our logical

Page table
The Page Table converts the page number in the virtual address into the physical location of the base.

Translation lookaside buffer
Normally we need to do two memory lookups to access the memory. We need to get the page table from memory, then use the page table to get our frame from memory.
The Translation Lookaside buffer (TLB) solves this by being a special fast look up cache especially for this situation. We first check the TLB and then if we can't find our page, we then have to check the page table.

Hierarchical page tables

We use this because a standard page table can get very huge very quickly. That makes it hard to contiguously allocate it in main memory.
We can have a multi-level page table. The top level page table can have entries with correspond to other page tables.

Hashed page tables, Inverted page tables--just basic concept

Hashed: Hash the virtual memory address into a page table containing a linked list of elements that contain the virtual page number and the value of the mapped page frame.

Ch.9: Virtual Memory

Basic concept

Virtual Memory allows for the full separation of user logical memory from physical memory. This means we can have a logical address space that is much larger than the physical memory space. To do this we load only a part of a program into memory at a time.

Sparse page tables

Demand paging

The idea of demand paging is to bring a page into memory only when it is needed. We therefore add to our page table a valid-invalid bit. If this bit is set to valid, then that means we have the page loaded into memory, and if the bit is set to invalid then that means we need to load the page into memory.

Page faults--details of basic mechanism

When we access the page table and try to access a page that is invalid, we get a page fault. A page fault means that the operating system needs to spend time loading that page into memory.

Instruction restarts

A page fault also means that the instruction we were trying to accomplish has to restart.

Page fault rates, calculating paging overhead

For $0 <= p <= 1$, where $p = 0$ means we never have a page fault and $p=1$ means every memory access leads to a page fault.

Estimated Access Time (EAT) = (1-p)*memory_access + p*(page fault overhead)

We expect that page fault overhead will be far larger than memory_access time. Therefore even a small amount of page faults will dramatically increase our EAT.

Page replacement strategies

Page replacement strategies answer the question "what happens when we run out of free frames?" If we have no free frames then we cannot successfully resolve a page fault, we cannot load that page into memory. Page replacement strategies pick a "victim" page to replace with the page we demand. This successfully completes the separation of logical memory from physical memory.

FIFO vs. Optimal vs. LRU

FIFO: First page in is the first page to be taken out

Optimal: Replace the page that will not be needed for the longest amount of time

LRU: Replace the page that has been least recently used

LRU approximations

LRU is hard to keep track of, so we have approximations, here are two:

Reference bit: Every entry in page table has a reference bit, initially set to 0. The bit is set to 1 if the page is referenced. We replace pages with reference bit = 0 first.

Other is second chance

Second chance algorithm

Page table has reference bit. We move through the table in FIFO. If the reference bit is set to 0 we replace it. If the reference bit is 1, set it to 0 and go to the next page.

Reference string-based evaluation

Best shown by example (included on practice exam here)

Allocation strategies

Each process needs a minimum number of frames given to it. Here are some strategies for allocating those frames.

Equal, proportional

Equal: Give each process an equal number of the total frames. So if we have 100 frames and 10 processes, each process gets 10 frames.

Proportional: Allocate according to the size of the node. If we have 99 frames and process 1 is size 10 and process 2 is size 20, then process 1 gets 33 frames and process 2 gets 66 frames.
Both of these are examples of fixed allocation.

Thrashing

What is it, when does it occur

Thrashing means a process has become busy just swapping pages in and out. Thrashing occurs when we have a high degree of multiprogramming. In this case the sum of the size of the localities becomes larger than the total memory size.

Working sets

Working sets are a way around this.
A working set is a fixed number of pages. We keep track of how many pages inside the working set each process references. If we add all of these up we get D, the total demand of frames. For m = total number of frames, the working set imposes a policy that if D > m then it will suspend or shut down another process.

<u>Ch.10: Storage Management</u>

Know the basic types of storage

- **Magnetic Disks**
- **Magnetic Tapes: Relatively permanent and holds large quantities of data but has very slow read speeds**
- **Tertiary: Low cost and generally removable media. Examples include floppies and CD-ROMs**
- **WORM: Stands for "Write Once Read Many Times." Very reliable and durable.**

Disk-based storage:

Disk layout (cylinder, head, sector)

Disk based storage has a read write head that scans a magnetic disk for data. It scans the cylinder for a sector of data.

Disk scheduling

FCFS, Shortest Seek Time First (SSTF), SCAN/C-SCAN, LOOK/C-LOOK

- **FCFS: First Come, First Serve. We simple serve the first request that comes in. Normally not a good algorithm**
- **SSTF: Shortest Seek Time First. Serve the request that takes the least amount of time for the head to get to.**

- **SCAN. Also called the Elevator algorithm because it goes up and down the disk like an elevator. This algorithm bounces from one end of the disk to the other, servicing all requests along the way.**
- **C-SCAN: Similar to SCAN except instead of bouncing to one side to the other, when this algorithm hits one side it immediately jumps back to the other side. It does not service any request when jumping from one end back to the other.**
- **LOOK/C-LOOK: Very similar to SCAN and C-SCAN respectively except instead of going to the ends of the disk, LOOK and C-LOOK are bounded by the location of the requests they receive.**

RAID: basic concept, plus know at general level:
Level 0: striping; Level 1: mirroring; Level 5: distributed parity
For each, what speed and reliability pros and cons are

- **RAID 0:**
    - **Uses data striping; treats multiple disks like one disk. This means we have fast write times but there is no reliability mode in place**
- **RAID 1:**
    - **Mirrors/copies data across multiple disks. So if you have 5 disks, you now have 5 copies of the same data. This is quite slow but extremely reliable**
- **RAID 5:**
    - **Uses Block Interleaved Parity. Basically a data block is split across multiple disks (interleaved), with a parity block. The parity block ensures that data is not lost, but it is not quite as quick as RAID 0.**

<u>Ch.11: File System Interface</u>
What is a file?
A file need only be a contiguous logical address space

File attributes
- **Name: Human readable name for file**
- **Identifier: Unique tag to identify a file**
- **Type**
- **Location**
- **Size**
- **Protections: Who can read, write and execute (when looking at UNIX)**
- **Time, date and user identification**

Access methods: Sequential vs. Direct

- **Sequential: Read and write data one piece after another**
- **Direct: Read/write a specific (directly stated) piece of data.**

Directories:

A directory is a collection of nodes with information about all files

Single level vs. two-level vs. multi-level tree vs.
directed acyclic graph (DAG) vs. general graph
-- Features of each type

- **Single level: There is one top level directory that is shared by all users. This has problems with naming conflicts and does not allow for grouping**
- **Two-level: Each user gets their own single level directory. This fixes the naming conflicts but still does not allow for grouping**
- **Tree Structure: There is a root directory with other child directories which reference other directories or files. This allows for efficient searching, clears up naming conflicts and allows for grouping**
- **Directed Acyclic Graph (DAG): Like Tree Structure but allows for shared subdirectories and files.**
- **General Graph: Like a DAG but can have cycles. Directories are now able to have references back to higher level directories. If we want to avoid cycles we will either have to disallow links to subdirectories, or implement a cycle detection algorithm.**

File system mounting

In UNIX when you mount a file system you first pick a directory to mount the file system into. Then we simply replace the node in the tree that corresponds to that directory with the top level node of the new file system. We can put the old node back in place when we unmount a directory.

UNIX protection (user, group, world RWX permissions)

In UNIX we can define read, write and execute (rwx respectively) privileges for the owner, the group the file belongs to, and the rest of the world
Reading corresponds to 4, writing to 2 and executing to 1.
So if we want to make so the owner can read, write and execute, the group can read and write and the rest of the world can only read we would do chmod 764 file_name
As 7 = 4 + 2 +1 and 6 = 4 + 2

Ch.12: File Sytem Implementation

Boot block, Volume control block, file control block

Boot block: Contains info need by the system to boot the OS. Normally is the first block of the volume.
Volume Control Block: Contains volume details like total number of blocks, number of free blocks, etc.
File Control Block: A per file block that contains details about the file. This includes the inode number, permissions, size, dates, etc.

Allocation methods:

Contiguous, Extent-based, Linked, and…

- **Contiguous: Every file occupies blocks that are physically next to each other (contiguous)**
    - **Simple and normally has the best read performance but can be hard to find space and leads to external fragmentation**
- **Extent-Based: allocate blocks in one or more contiguous block of disks called extents**

Indexed allocation:

Each file has its own index block(s) of pointers to its data blocks.

Index block

An index block contains pointers to data blocks

Direct and indirect pointers

A direct pointer directly points to a data block, while an indirect pointer doesn't. For example in indexed allocate we point to an entry in the index table which then points to our data block.

The UNIX inode multi-level block index

We can have multiple level of block indexing. A logical address can go into an index table which then goes into another index table which then maps into the physical block. This would an example of a double index. Multi-level indexes let us map from a small amount of data blocks into a large amount of data blocks.

Free block list representations

Bitmap, linked list, UNIX's hybrid linked list (mentioned in lecture)

Bitmap: Have a bit map that represents whether a block is free or open. Bitmap[i] = 1 if free or bitmap[i] = 0 if not free. The issue is that this takes up extra space

Linked List: Each free block has a reference to the next free block, we keep a pointer to the head of this linked list. This makes it harder to access

contiguous blocks, but doesn't take up extra space.

<u>Ch.13: I/O systems</u>

Multi-bus architecture model

I/O control: I/O ports and special cmds, vs. memory-mapped I/O

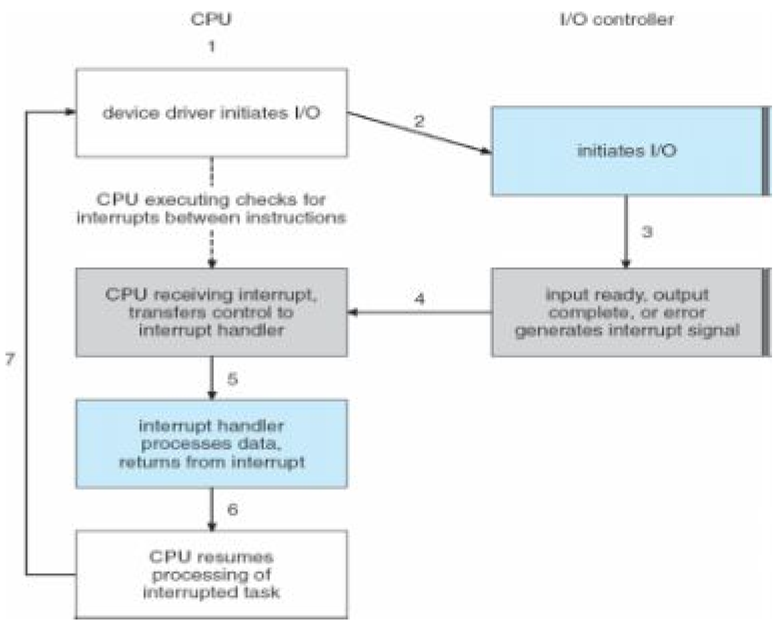
Port-mapped I/O: uses special class of CPU instructions for performing I/O. These instructions can read and write one to four bytes to an I/O device. I/O devices have a separate address space from general memory.

Memory-mapped I/O: I/O device is accessed like it is part of the memory. Load and Store commands are executed for reading from and writing to I/O device. I/O devices use the same address bus as memory, meaning that CPU can refer to memory or I/O devices based on the value of the address.

I/O control models:

- Polled: **process where the controlling device waits for an external device to check for its readiness or state.**
  - **Read busy bit from status register until 0**
  - **Host sets read or write bit and if write, copies data into data-out register**
  - **Host sets command-ready bit**
  - **Controller sets busy bit, executes transfer**
  - **Controller clears busy bit, error bit, command-ready bit when transfer done**
  - **Busy-wait polling: computer waits until the device is ready**

- Interrupt-driven


CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

- DMA (Direct Memory Access):
  - **A method that allows an I/O device to send or receive data directly to or from the main memory, bypassing CPU to speed up memory operations.**
  - **Requires DMA controller**
  - **OS writes DMA to transfer block into memory**
    - **- Source and destination addresses**
    - **-Read or write mode**
    - **-Count of bytes**
    - **-Writes location of command block to DMA controller**
    - **-Bus mastering of DMA controller**
    - **-When done, interrupts to signal completion**

Block vs. character devices

Block devices: the driver communicates by sending entire blocks of data.

- **Commands include read, write, seek**
- **Raw I/O, direct I/O, or file system access**
- **Memory-mapped file access is possible**
- **DMA**

Character devices: the driver communicates by sending and receiving single characters (bytes, octets).

- **Commands include get() and put()**
- **Libraries layered on top allow line editing**

Non-blocking and asynchronous I/O

Non-blocking: I/O call returns as much as available. I/O request is queued straight away and the function returns.

- **User interface, data copy**
- **Implemented via multi-threading**
- **Returns quickly with count of bytes read or written**
- **select() to find if data ready then read() or write() to transfer**

Asynchronous: process runs while I/O executes. Permits other processing to continue before the transmission has finished.

- **I/O subsystem signals process when I/O completed**

Ch.14: Protection

General knowledge of concepts:

Principle of least privilege

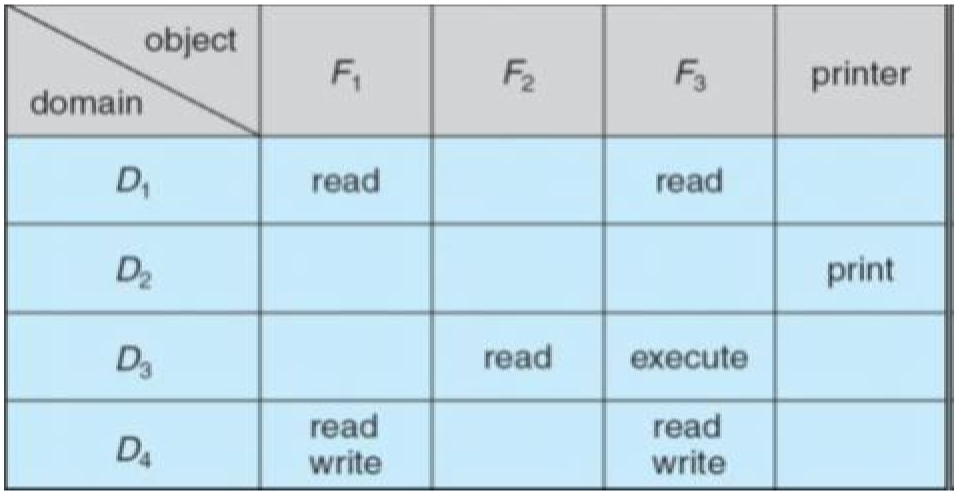
The principle of least privilege says that we should give programs, users and system just enough privileges to complete their tasks. If a process only needed to write and read to an object, if we also gave it the ability to execute that object then we would be violating the principle of least privilege.

Domain of protection

The domain of protection is a set of access rights expressed as <object-name, rights-set>

Access matrix

An access matrix is a matrix that specifies access rights for objects and domains. Columns are objects and rows are domains. Access(i,j) is the set of operations that a process in domain i can invoke on object j.



object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Implementation: access lists for objects vs. Capability lists for domains

- **Access List: This is a column focused implementation of the Access Matrix. We have each column be an access list list for an object. This says how an object can be accessed in each domain.**
- **Capability List: This is a row focused implementation of the Access Matrix. We have each row be a capability list. These lists tell us what we are capable of invoking on objects within a domain.**

Ch.15: Security (only if covered in class)

General knowledge of concepts:

Basic security problem definition

System secure if resources used and accessed as intended under all circumstances; however this is unachievable. Defense is harder than offense, and it's easier to protect against accidental misuse than malicious misuse

Man in the middle

Intruder sits in data flow, masquerading as sender to receiver and vice versa

Social engineering

Social engineering is an attack vector that relies heavily on human interaction and often involves tricking people into breaking normal security procedures. Think of phishing or dumpster diving.

Trojan horse

Code segment that misuses its environment Exploits mechanisms for allowing programs written by users to be executed by other users. Examples include: spyware, pop-up browser windows, covert channels.

Trap door

Specific user identifier or password that circumvents normal security procedures. Could be included in a compiler. Have to analyze all source code for all components of a system in order to detect trap doors.

Buffer overflow attack

Most common way for an attacker outside the system on network or dial-up connection to gain unauthorized access. Authorized user may also use this exploit for privilege escalation. Attack exploits a bug in a program where attacker sends more data than the program was expecting.

Port Scanning
Automated attempt to connect to a range of ports on one or a range of IP addresses. Frequently launched from zombie systems (previously compromised, independent systems that are serving their owners while being used for nefarious purposes) because port scans are detectable.

Denial of Service
Overload the targeted computer preventing it from doing any useful work.

Cryptography:
Means to constrain potential senders (sources) and/or receivers (destinations) of messages.

- **Based on secrets (keys).**
- **Enables:**
  - **Confirmation of source**
  - **Receipt only by certain destination**
  - **Trust relationship between sender and receiver**

Symmetric encryption: what is it?

Same key is used to encrypt and to decrypt

Key exchange problem

How do we exchange whatever keys or other information are needed so that no one else can obtain a copy?

Asymetric (public key) encryption: what is it?

There are different encryption and decryption keys. An entity preparing to receive encrypted communication creates two keys and makes one of them (public key) available to anyone who wants it. Therefore, any sender can use that key to encrypt, but only key creator can decrypt (using private key).

How is it used?

How can it be used for authentication

What is key distribution problem?

Need proof of who (or what) owns a public key so we don't make the mistake of encrypting our message with malicious user's public key. We solve this problem with digital certificates.

Digital certificate: public key digitally signed by a trusted party. THis trusted party receives proof of identification from some entity and certifies that the public key belongs to that entity