Exam 1 Guide

# Chapter 1 & 2: Introduction

## History of Processor Sharing

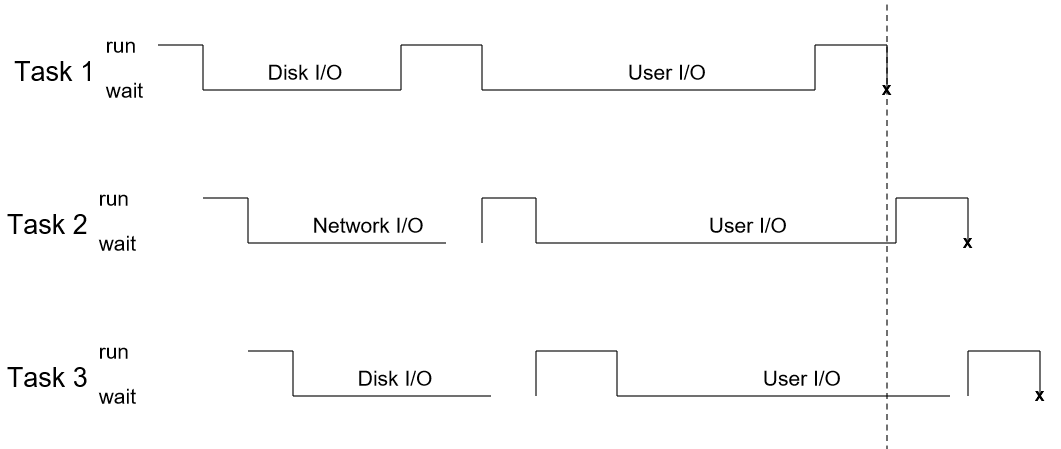
### Stage 0: DIY Processor Sharing

* + The person is the operating system!
    - (they manually load each program)
    - They have to switch out the programs

### Stage 1: Batch processing

* + You have a queue that you place tasks in
  + The monitor takes the place of the person
    - List of processes shoop de boop the grabbing is automated and it gets stuff
  + Gets control of process, gives control back etc etc
  + You don’t really have protection… only one process at a time
  + Bad communication too!

### Stage 2: Multiprogramming

* + We want to keep the CPU as busy as possible!
  + 
  + IO Blocking! Another process can use the CPU while another is blocking on IO

### **Time sharing (something added to multiprogramming to make it better):** everyone gets a certain amount of time to run on the CPU

* + - It doesn’t matter if you ain’t done, you gotta freeze and then come back to it
    - *Problem:* switching between tasks takes time (context switchingggggg)
* **Memory mapping (used whenever you context switch) -** we have to store the program for later so this means we are remembering the states
  + We still don’t have protection, because we still can’t control who can do what and when
* Notification (interrupts) is run by a watchdog program - it’s just overhead. This is a PROBLEMO COMPRENDO
* Boop

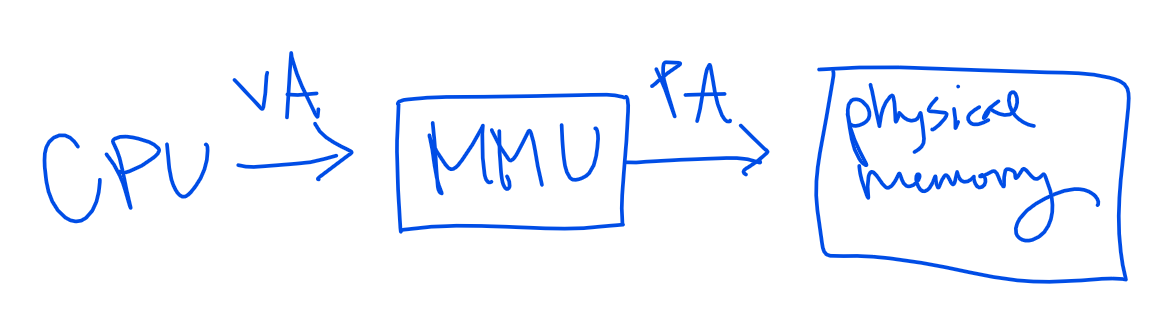
### Von Neumann - an algorithm for executing programs

* + Repeated:
    - Fetch
    - Decode
    - Execute (write results)
* Now we’re going to “**pipeline**” to **parallelize** Von Neumann
  + Pipelining: rearranging the instructions
* **Superscalar**: multiple instructions simultaneously executing on a single processor
* multi/hyperthreading - taking what we’ve talked about and putting it on multiple threads
  + Supa fast context switching
  + Simulating a multicore processor like a boss
* **DIFF b/w multiprocessing and multiprogramming**
  + Multiprogramming is on ONE processor

### Process of Context Switching

* + Saving the state (like registers) for one task and
  + Restoring the state of the next task

### Sharing Memory!

* + **Problem to address**: To allocate memory, we have to give physical memory, but we don’t want people accessing each other’s memory
  + 
  + Denton is ranked 330th at baylor
* Idea behind a VM
  + It’s gonna think it’s the only OS on the computer
  + 
  + The kernel is always running
* Multiplexing… WE DIDN’T TALK ABOUT IT
* Dual Mode! We have a dual layer system that handles dangerous stuff
  + Kernel mode is in the Danger Zone
  + User mode is like a padded room
  + If we get an interrupt (like in a system call) we can switch back and forth
* **Trap:** specifically software generated interrupt either by error or requested (system call)

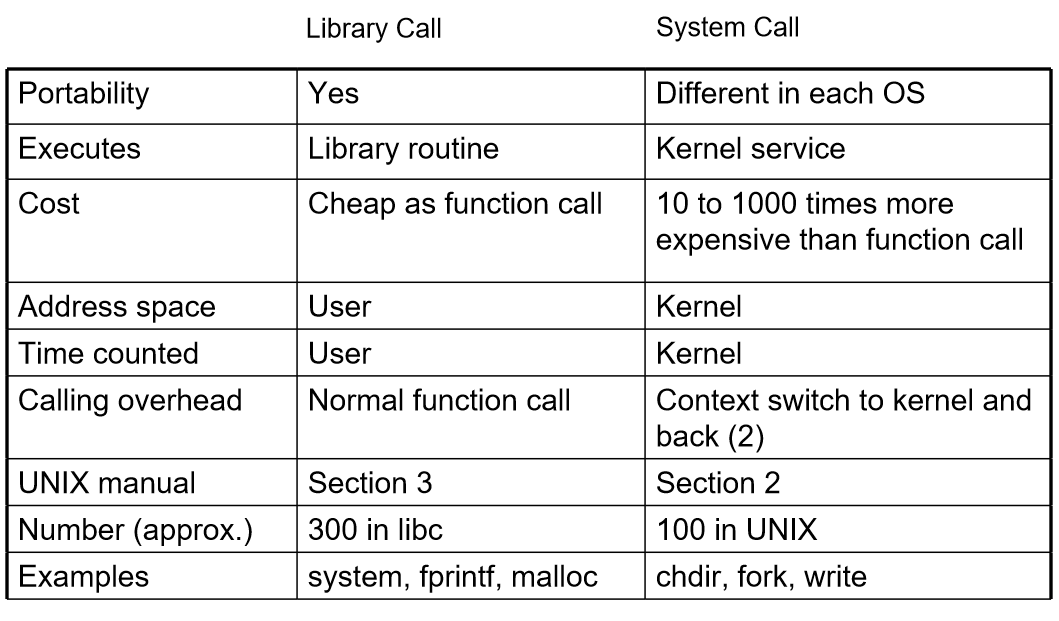
### Handling interrupts

* + Interrupts can’t be interrupted
  + After handler completes, interrupted task resumes where it left off

### We Need An Operating System. We want…

* + Usability and Portability
    - Resource abstraction/interface
  + Efficiency
    - Resource management
  + Control
    - permissions
  + Isolation
    - Virtualizations
* System callzzzzz
  + Use kernel mode permissions and they can do stuff
  + OS specific

### Library vs system calls

* + Library calls follow an OS-independent standard
  + Usability
    - System calls can often be more detailed and difficult to use
  + 
* Kernel is running all da time

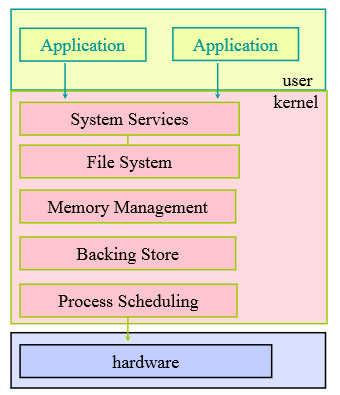
### Monolithic Kernels

* + Earliest and most common OS architecture
  + Everything is contained in the kernel
  + V efficient
  + Susceptible to malicious code

### MS DOS

* + Batch processing

### Layered Approach

* + Divides OS into a # of layers
  + Each layer **only uses stuff under it**
  + Library calls will only use their stuff or system calls
  + 

### Microkernel (opposite of monolithic kernel)

* + Separate kernel programs into system and user level programs
  + Moves as much from the kernel into “user” space
  + Soooo much context switching
  + Advantages
    - Easier to extend microkernel
    - Easier to port the operating system to new architectures
    - More reliable
    - Fewer failures
  + Disadvantages
    - Slow slow slow - performance overhead of user space to kernel space

### Modules

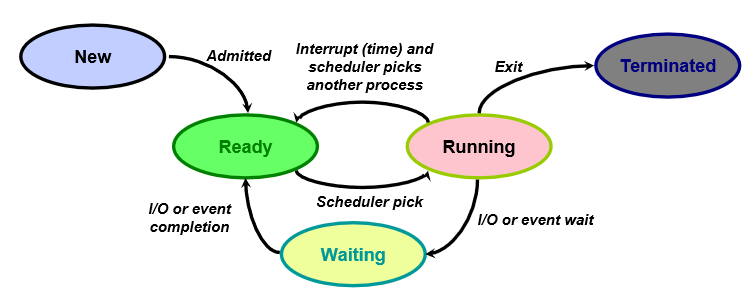
* + Your components are objects
  + Layers, but not dependent

# Chapter 3 - Processes

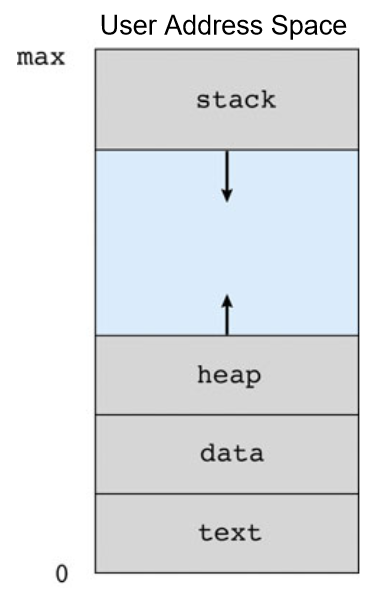
### Multiprogramming

* + Want…
    - Multiple tasks “simultaneously” executing
    - To handle asynchronous events
  + Hey what if…. Each program has its own virtual machine?!?!?!?!! Omg
  + **What’s a process? A program in execution. Bruh.**

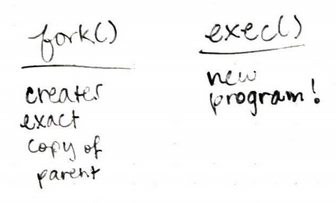
### Process States

* + States:
    - New - the process is being created
    - Running - instructions are being executed
    - Ready - process is waiting to be assigned to a processor
    - Terminated - the process has finished execution
    - Waiting - the process is waiting for some event to occur
    - 
  + Transitions! Woot
    - **Why would a process change state?**
      * Program action (system call)
      * OS action (scheduling decision)
      * External action (interrupt)
* Execution states and whatnot
* Boop
* Oh
* Party

### Process Control Block (PCB)

* + Linux stores errything in a PCB
  + “Did you just assume my language?”
* Secret Process Autopsy
  + Text (code)
  + Data (global vars)
  + Heap (dynamic mem)
  + Stack (function params)
    - Return addr
    - Local var
  + 
* **GANTT CHART**

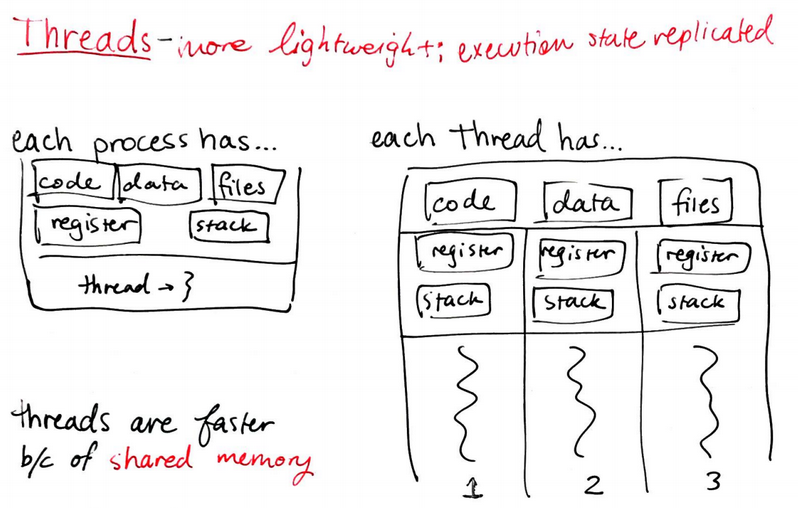
### Process Life Cycle - UNIX

* + PID 0 is usually the scheduler process
  + Init - mother of all processes
  + Getty - login process that manages login sessions
* 
* Process Termination
  + Process executes last statement and then asks OS to delete it by using exit()
  + Parent may terminate execution of children
* Independent process cannot affect or be affected by other processes.
* Cooperating processes can “ “
* IPC = interprocess communication

### COMMUNICATION MODELS

* Shared memory model
  + Faster, but more dangerous
* Message passing model
  + Info to kernel then to other process
  + Pipes (named are bidirectional, unnamed aren’t)
* Synchronous Communication
  + Must wait for the receiver to receive the stuffs
  + “Legitimate form of communication”
* Asynchronous Communication
  + Don’t need to wait for the receiver to receive the stuffs
* Blocking
  + Completion is dependent on certain events
  + “Paradigm”
* Non-blocking
  + Duh

# **Chapter 4: Threads**



Kernel doesn’t know how many threads you have in the process

* Boop

Thrashing: context switching more than actually doing crap

fork(): Copy all threads in process or just calling thread?

* OS Dependent

exec(): Usually replaces the entire process

Synchronous (SIGSEGV) Asynchronous (SIGINT)

Thread Pools: a fixed-size set of threads

# Chapter 6: CPU Scheduling

Preemptable Resources:

* Can be forcibly removed (and maybe returned) without ill effects

Non-preemptable Resource:

* Duh

Resource Management Tasks:

Scheduling

Allocation

Short-term scheduling (CPU scheduler): who gets to play this game

Long-term scheduling: who is on the team for the season

Long-term scheduler controls degree of multiprogramming

**Utilization**: keep CPU busy

**Throughput**: completed processes per unit time

*Turnaround time*: time between job arrival and finish

*Response time*: time from request to first output

*Waiting time*: time in the waiting queue

**Fairness**: all jobs get the same amount of CPU overtime

**Overhead**: reduce number of context switches

*Variance*: variability in response time

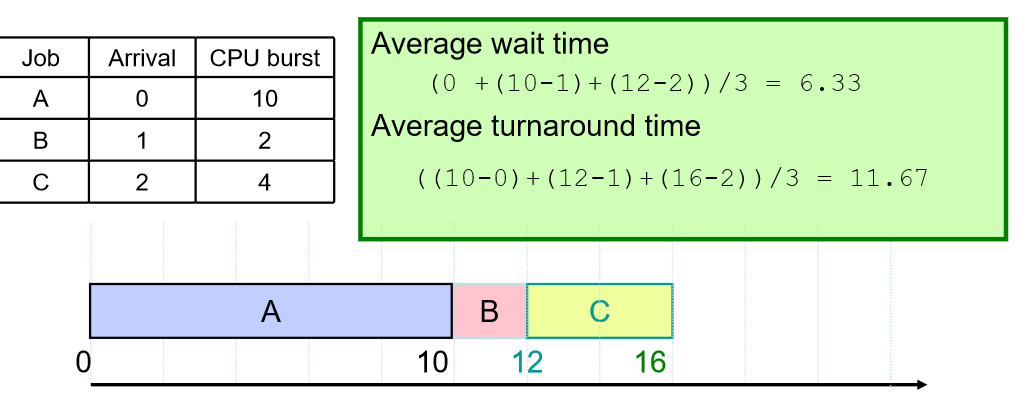
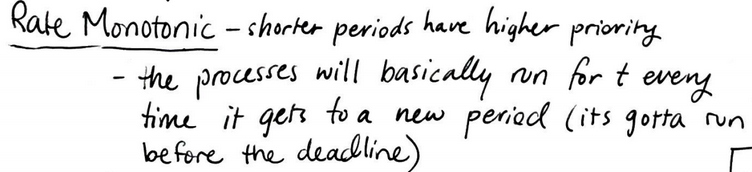
*User*

**Scheduler**

Performance criteria depends on system

* All systems
  + Fairness
  + Overall system utilization
  + Policy enforcement
* Other stuff

## Algorithms for Short Term Scheduling (using Gantt chart)

* FCFS (First come first serve)
  + Idea: maintain fifo list of jobs as they arrive
    - Non-preemptive policy
  + Doesn’t work on timesharing systems
  + 
  + Simple implementation
  + Pretty fair
* SJF (shortest job first)
  + Incorporates FCFS if the jobs are the same length
  + Non-preemptive
  + Advantages
    - Optimal for minimizing average wait time
    - Keeps io busy
  + Disadvantages
    - Starvation for long jobs
* STCF/SCTF (shortest time to completion first)
  + Idea: add *preemption* to SJF
  + If a new process comes that’s shorter, it’ll kick the current process off and go ahead and run
  + Long jobs have a hard knocked life
* Round Robin (RR)
  + Interactivity in time-sharing
  + FIFO + timesharing
  + Run each job and move to back of FIFO queue
* Priority Based
  + Each job is assigned a priority
  + Schedule the highest priority job
  + Lowest priority jobs starve
* Multilevel Queue Scheduling
* **Real-Time CPU Scheduling**
  + 3 characteristics of process
    - Period p - need CPU every t time units
    - Deadline d - need to be completed by this time
    - Processing time
    - What is their ordering?
    - Rate - t/p
  + Time < deadline < period
  + 3 algorithms
  + Rate Monotonic
    - 
  + Earliest Deadline First
    - Only switch if the new process has a sooner deadline
    - But context switching takes time
  + Proportional Sharing
    - I have some shares but if I don’t have enough shares then you gotta wait

# Chapter 5: Synchronization

Why synchronization? No concurrent writing.

* As much interweaving as possible without compromising data

Threads don’t share local variables, but they share anything else

Race condition - two things are racing to edit a variable (hit critical section at same time)

* Non-deterministic - can be hard to find
* Time-dependent
* Avoid race conditions - no reading and writing at the same time
  + Critical section - part of the program where the shared resource is accessed
  + Can run into deadlock or starvation if not careful
* Required properties of critical section
  + Mutual exclusion - only one thing in CS at a time
  + Progress - can’t stop other things from waiting
  + Bounded waiting - you eventually have to go (no starving!)
  + Should be atomic operations

Synchronization solutions

* Software solutions
  + Shared lock - lock the door behind you and other person waits
    - Problem - if you both walk in at the same time, you’re in trouble (both threads read lock as unlock)
    - No check then set
    - Want it to be in state of intent
  + Strict alternation - stand in line for room
    - Problem - person in line may not need thread, and threads behind them starve
  + Check state then lock - array of locks, check array before going in
    - Problem - both threads can set their locks after checking
  + Lock then check - array of locks, lock before checking
    - Problem - both threads set locks then wait on the other
  + Defer, back off lock - add deferral and delay
    - Problem - Canadians (you go first! no, you go first!)
  + Peterson’s Algorithm - not best solution, but prevents deadlock (2 threads)
    - No longer looking at state of the door
    - Check if it’s your turn or if the other person is not using the room
* Hardware solutions
  + Atomic operations (testAndSet, swap)
  + Bounded waiting algorithm (use testAndSet - switching values and returning old value)
* Synchronization layering (based on hardware operations)
  + Locks - only provide mutual exclusion
    - Allocate, acquire, release
  + Spin locks - busy waits (wait over and over again)
  + Synchronized block (Java)
    - Every Java object has a lock (any object can be the lock)
    - Have to own the object to use it
* Problems
  + System might try to help you
  + We need more than atomicity
  + Volatile variables - forces everything to happen or not happen

Semaphores - locks++

* Maintain resource count, keep (default non-deterministic) queue of waiting threads
* To enter CS, acquire semaphore permit
* To leave CS, release the permit
* Semaphore operations
  + Allocate
  + Acquire
  + Release
* Types
  + Binary semaphore - 1 permits
  + General (counting) semaphore - n permits
* Problems
  + No bounded waiting if non-deterministic
  + Deadlock if waiting on threads that wait on you
  + Prone to bugs, difficult to program
* Versatile

Synchronization Case Studies

* Producer/Consumer
  + Producer puts into buffer
  + Consumer gets from buffer
  + Solution: 3 semaphores
    - Mutex semaphore: keep from reading/writing to same place
    - Consumer semaphore: make sure it’s not empty
    - Producer semaphore: make sure it’s not full
* Reader/Writer
  + Multiple threads trying to read and write at the same time
  + Need to make sure things don’t screw up
  + Reader priority - readers get to go before writers
  + Writer priority - writers get to go before readers
* Dining Philosophers
  + 5 philosophers, 5 chopsticks
  + Need 2 chopsticks to eat
  + Array of chopstick semaphores
  + Solutions
    - Considerate - people starve
    - Selfish - only one person gets to eat at a time
    - “Enfoodened” - 3 states (hungry, eating, thinking)
      * Check to see if chopsticks aren’t in use (neighbors are thinking)
      * “Liveness” - make as many good things happen as possible
        + When done, see if neighbors want to eat

Monitors - figment of your imagination

* Collection of functions and state grouped together
* Mutex by default (only allow one thread to be in the monitor at a time)
* How do threads block? Condition variables (not implemented in Java - use while loop)
* Java monitor - objects have intrinsic locks
  + wait - releases lock
  + notify - wake up single thread (non-deterministic)
    - Use if only one thread can possibly be woken up or if only one thread should come next
    - If no threads are waiting, calling notify does nothing
      * Only allow someone to notify if someone needs to be notified
  + notifyAll - wake up all threads (only one can proceed)
* Java monitors are not pure monitors (methods aren’t technically synchronized, anybody can use lock (not just monitor), fields not necessarily private, single conditional variable)

Message passing - send message, wait for it to come back

* Alternative to semaphores and monitors

Example

* Monitor is public toilet (lock behind them)
* Semaphore is bike hire place (certain number of bikes, gotta wait for someone to return)

# Chapter 7: Deadlock

Deadlock - set of processes where each is waiting for an event that only another process in the set can cause

* Focused on resource sharing

Resource types

* Preemptable - can be taken away
* Non-preemptable - can’t take it away

Use: request, use, release

Resource Allocation Graph

* Vertices: process (circle), resource (square)
* Edges
  + Request: process to resource
  + Assignment: resource to process

Necessary conditions for deadlock

* Mutual exclusion - one process can use a resource at a time
* Hold & wait - process can hold resource while waiting for another resource
* No preemption - stuff you’ve given can’t be taken away
* Circular wait - chain of processes waiting on other processes that forms a cycle

Handling Deadlock

* Ignore (cheaper & easier)
* Deadlock prevention
  + Eliminate mutual exclusion - make resources sharable
    - Problem - don’t want simultaneous r/w
    - Don’t do this
  + Eliminate hold & wait - two approaches
    - Get all resources at once (one lock gets everything)
      * Problems: May not need everything, holds everything longer than necessary
    - Only hold one resources at a time
      * Problems: Might need multiple resources, can starve (can’t use it)
  + Eliminate no preemption - two approaches
    - Preempt requester - drop stuff to get stuff
    - Preempt holder - force another process to drop
    - Problem: not possible if resource can’t be saved & restored
  + Eliminate circular wait - impose a global ordering
    - Problem: inefficient (might do things in wrong order)
  + Problem may line up, but don’t force it
* Deadlock avoidance - looking at future behavior
  + Dynamic decision based on current and potential maximum resource allocation
  + Process Resource Trajectories (check PPT)
  + Safe state - is there ANY way out? (Even if everyone requests their max)
  + Unsafe state - there’s no way out
  + Banker’s Algorithm
    - Assumptions
      * Assumes processes declare max potential needs ahead of time
      * Assumes single resource (multiple units), but can work with multiple resources
    - Algorithm
      * System will only grant process request if it doesn’t lead to an unsafe state
      * Deals with max resources and current resources
* Deadlock Detection & Recovery - allow deadlock, then fix it
  + Detection
    - Single resource - check periodically for cycle in a graph - O(n^2)
    - Multiple resources - if there’s no cycle, there’s no deadlock, but vice versa isn’t necessarily true
    - DIFFERENT FROM BANKER’S
      * Finding out if you’re already deadlocked
      * Can pull resources to Available from Current to make it work
  + Recovery
    - Process termination - abort deadlocked processes and reclaim resources
      * Abort 1 at a time until we fix it
      * Can be based on low-priority, how long it’s been executing, how many resources it holds, batch or interactive, # of processes that have to be terminated
    - Resource preemption - get resources back
      * Issues: victim selection, rollback (save checkpoints), starvation
    - Issue: how often should it run?
      * After every request, periodically, when CPU utilization is low, when we suspect deadlock (inactive thread)

# Chapter 8: Memory Management

Memory is sharable. We should do that.

Methods

* Static relocation - allow transparent sharing
  + Process can be anywhere in memory
    - OS finds free space
    - Modifies addresses statically
  + Allows multiple processes to run, no hardware support
  + No protection, need contiguous address space
* Static address binding
  + Can compile the physical address at different times (compile, load, execution)
* Address Space
  + Don’t want physical addresses - privacy (no hacking), bounded violation
  + Logical address space - starts at 0 (per process)
  + System has to map between them
* Dynamic relocation - protect processes from each other
  + Needs MMU (Memory Management Unit) to convert between physical and logical addresses
  + Protection by address range testing
* Memory Overload
  + Not enough memory to hold all processes waiting - put in waiting room (backing store)
  + Problems
    - Swapping is expensive
    - Have to swap before you can be given the processor
    - Either don’t swap or have an OS buffer
* Swapping in - how do we allocate?
* Contiguous memory allocation
  + Fixed-size partition
    - Divide memory into fixed-size chunks that new processes get
    - Problems
      * Partition size (Goldilocks - can’t be too big or small)
      * Internal fragmentation
  + Fragmentation
    - Internal - unused memory in a partition
    - External - unused memory between partitions
  + Variable-size partition - process requests sides that it needs
    - Placement depends on availability, and exits make holes
    - OS decides where things go and tries to combine holes
    - Keep list of available holes
    - 3 ways to place
      * First-fit - find the first hole big enough
        + 50% rule - for N blocks, N/2 blocks are wasted
        + Compaction - relocation to eliminate fragments
        + Leaves a bunch of average-sized holes
      * Best-fit - find smallest hole big enough
        + Leaves variety of holes
      * Worst-fit - find biggest hole big enough
        + Leave a few big holes
      * Buddy allocation (Linux) - allocating blocks of size 2^N
        + Divide and combine to fit processes
  + Summary
    - Fast, simple, inexpensive, but has problems (have trouble getting size and place right, and can’t share limited parts of address space)
* Paging
  + Divide process into chunks to be stored
    - Logical addresses have to be contiguous, but physical memory can be split up
  + Chunk size: 2^N
  + Page types
    - Physical page - frame
    - Logical page - page
  + Page address translation (logical to physical)
    - High-order bits - page number (p)
    - Low-order bits - offset in page (d)
  + Frame allocation - where in memory do the frames go?
  + Cache - temporary storage for recently allocated data (don’t send back yet)
  + TLB (Translation Look-Aside Buffer) - hardware cache for pages
    - Normal page table access - access page table, then physical memory (2 accesses)
    - Contains recently used page table entries
    - TLB Miss - not there (have to make the accesses)
    - Why it works
      * Temporal locality - using same thing over and over again
      * Spacial locality - using things close to one another
    - Page size
      * Bigger page - have entire page in memory
      * Smaller page - decrease internal fragmentation, but more pages (larger page table, fewer hits)
  + Advantages: No external fragmentation, can share address space
  + Disadvantages: internal fragmentation, page table memory reference
  + Extensions
    - Fixing large page tables (each has advantages and disadvantages)
      * Two-level page tables (page the page table)
        + This method solves the large quantity of page tables problem
      * Hashed page table (have virtual page number)
        + Easier and faster access to by using a hash function
        + However, this method can get pretty complex to implement because of collisions
      * Inverted page table (fitting everything into one page table)
        + Easy to organize
        + Searching through a single page table can be slow and inefficient
      * Juiced page table (Control frame access permission)
        + Not 100% about this
        + We give the frequently used frame a valid bit
* Segmentation