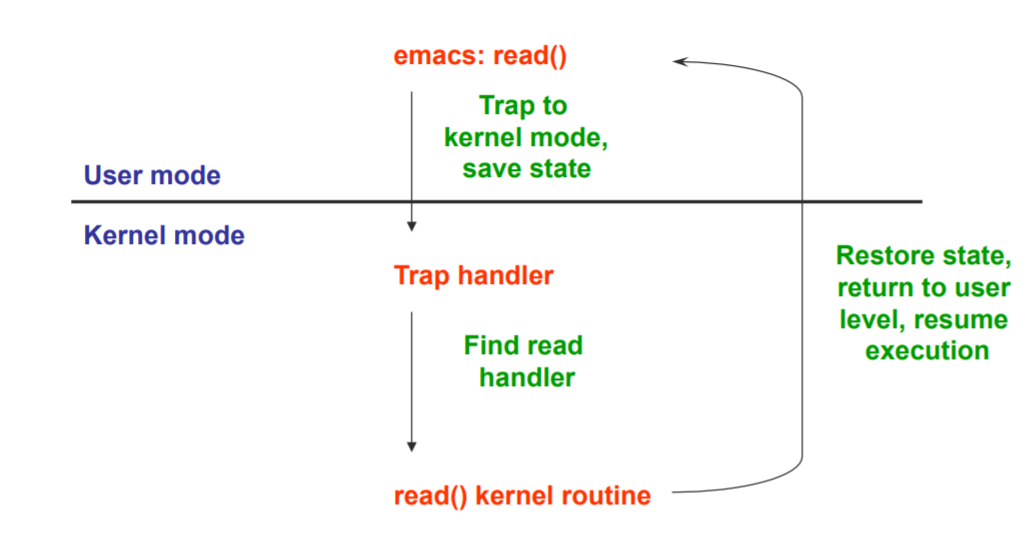
OS: program that manage computer hardware

* Exclusive access to hardware and data structure
* Directly access I/O devices
* Manipulate memory
* Manipulate registers
  + Kernel mode
  + Interrupt level

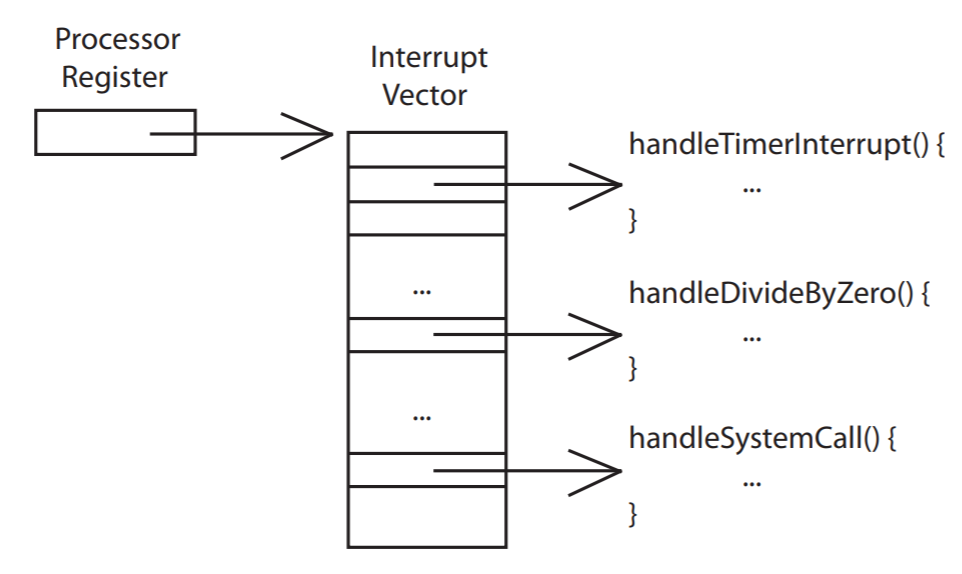
Privilege mode:

* OS (kernel) mode
  + OS executes in kernel mode
* User mode
  + User programs execute in user mode
  + Attempts to execute in user mode **trap** to OS mode
* Going from higher privilege to lower privilege
  + Special instruction to change mode
    - System calls
      * OS check syscall request and honor if safe
      * For user to do something privileged(I/O)
        + Crossing protection boundary or protected procedure call
      * Causing exception that invokes kernel handler
        + Passes parameter determine system routine

Vector table

* + - * Saves state that can be restore
      * 
    - Invokes designated handler
      * Jump to privileged code
    - Some kind of events happen in the system

Events:

* Unnatural change
* Immediately stop current execution
* Change mode, machine state, or both
* Handler
  + Event handler always execute in kernel
    - 
  + OS is one big event handler
* Two type of events
  + Asynchronous
    - Not Sync with CPU ticks
    - External event
    - Interrupts
      * I/O hardware interrupt
        + O maintains a vector table containing a list of addresses of kerel routines to handle various events

Ethernet receives packt, writes into memory

Signal interrupt

Stop current operation, switch to kernel saved machine state

CPU read vector table index by interrupt number, branches to address(ethernet device driver)

Process packet(read device registers to find packet in memory)

Restore save state from stack

* + - * Software and hardware timers
      * Types of interrupt
        + Precise

CPU transfer control only on instruction boundaries

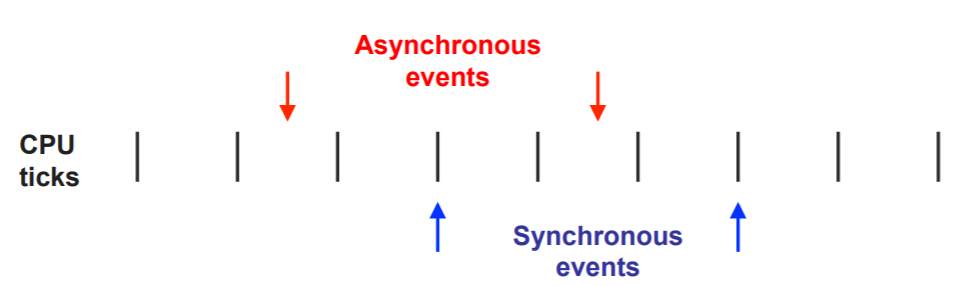
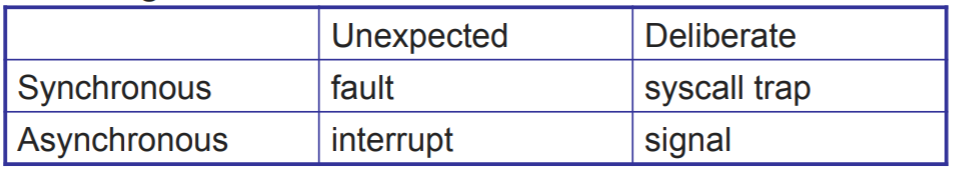
* + - * + Imprecise

CPU transfers control in the middle of instruction execution

* + Synchronous
    - sync with cpu
    - Executing instruction
    - Fault
      * unexpected event
        + Divide by zero
        + Access violation
      * Must save state(PC,reg,mode) so faulting process can be restarted
      * Registered handler
      * Kernel may handle unrecoverable fault by killing user process
        + No registered handler
      * Fault in kernel
        + Dereference NULL, divide by zero, undefined instruction

fatal error that causes OS crashes

Kernel halted, state dumped to core file, machine locked

* + 
  + 

Abstraction: defines a set of logical resources

Virtualization: isolate

Batch system: computer expensive, people cheap

Sleeping beauty model: os only work when needed

* Controlled direct execution
* Response to events
* Only OS can manipulate hardware or critical system state

CPU:

* manipulate device registers
* controlling access
* Interrupts, exception, system call
* Respond to external events
* Requires software to handle fault or trap
* Handle concurrency, isolation, virtualization

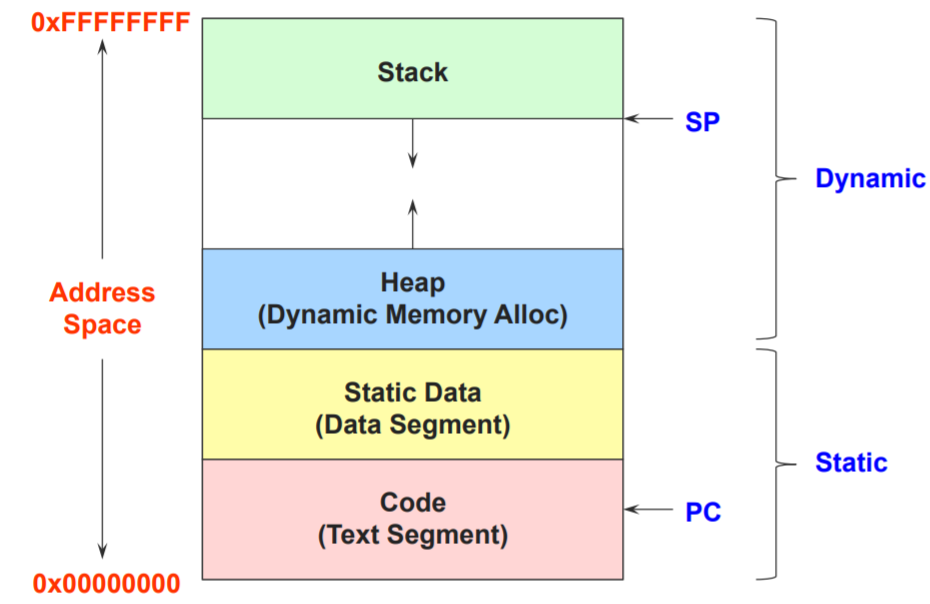
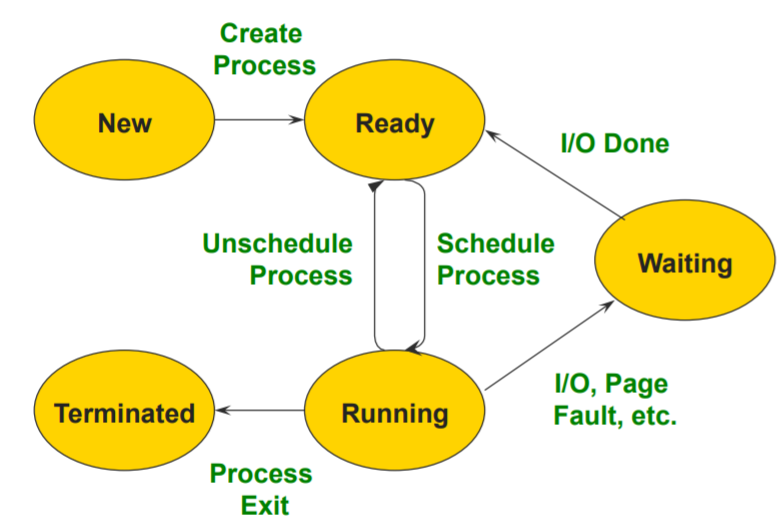
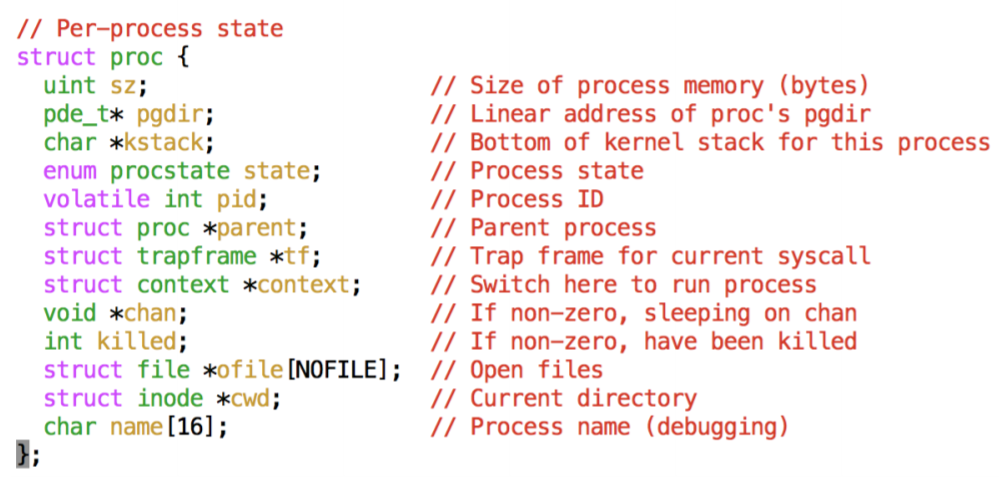
Fork()

* It copy the parent process to the child process
  + In the parent process
    - fork() returns the child processID
  + In the child process
    - fork() returns 0
* We don’t know the order of the process unless we use waitpid or wait() in parents

Timer

* Key for time sharing OS

Process

* OS abstraction for execution
* Unit of execution
* Contains all state for a program in execution
  + Static memory
    - Code and input data for executing program
  + Dynamic memory
    - Memory allocated by the executing program
    - 
  + Control registers
    - Program counter (PC)
  + General purpose registers with current values
  + Operating system resources
    - Open files, network connections etc
* A process is named using its process ID (PID)
* Switching process
  + Context switching
    - costy
* Execution state indicate what is doing
  + Running
    - Executing instruction on the CPU
  + Ready
    - Ready to be assigned to CPU
  + Waiting
    - Waiting for an event
  + 
* OS data structure represent each process is called process control Block(PCB)
  + PID
  + Execution state
  + Hardware state
  + Memory management
  + Scheduling
  + 
* Use queue to keep track of processes
* Creating new process is costly because new address space and data structure must b allocated and initialized
* Communicating between processes is costly because most communication goes through the OS
  + Inter Process Communication (IPC)
  + Overhead of system call and copying data

Process Creation:

* Unix: Fork()
  + Create and initialize new PCB
  + Create new address space
  + Initialize address space with copy of entire content of parent
  + Initialize kernel resources to point to resources uses by parent
  + Place PCB on the ready queue
  + Fork returns twice
    - Child PID to parent, 0 to child
  + process user ID is inherited
  + Parent either wait or continue in parallel(or both)
  + Useful when child cooperating with parent
  + Relies on parent’s data
* Windows: create process: Bool CreateProcess(char \*prog, char\* args)
  + Initialize new PCB
  + Initialize new address space
  + Load program
  + Copy args into memory in address space
  + Initialize saved hardware context
  + Place PCB on ready queue

Exec: int exec(char\* prog, char \*argv[])

* fork () is used to create program
* exec() is used to start a program
  + Stop current process
  + Load the program into process address space
  + Initialize hardware context and args for the new program
  + Place PCB on the ready queue

Wait: pause until a child process has finished

* wait()
  + Suspend current process until child process end
* waitpid()
  + Suspends until the specific child process end

Exit: exit(int status) : Free resources and terminate

* Terminate all threads
* Close open files, network connection
* Allocate memory
* Remove PCB from kernel

Thread : execution state: sequential execution stream within a process (PC, SP, registers)

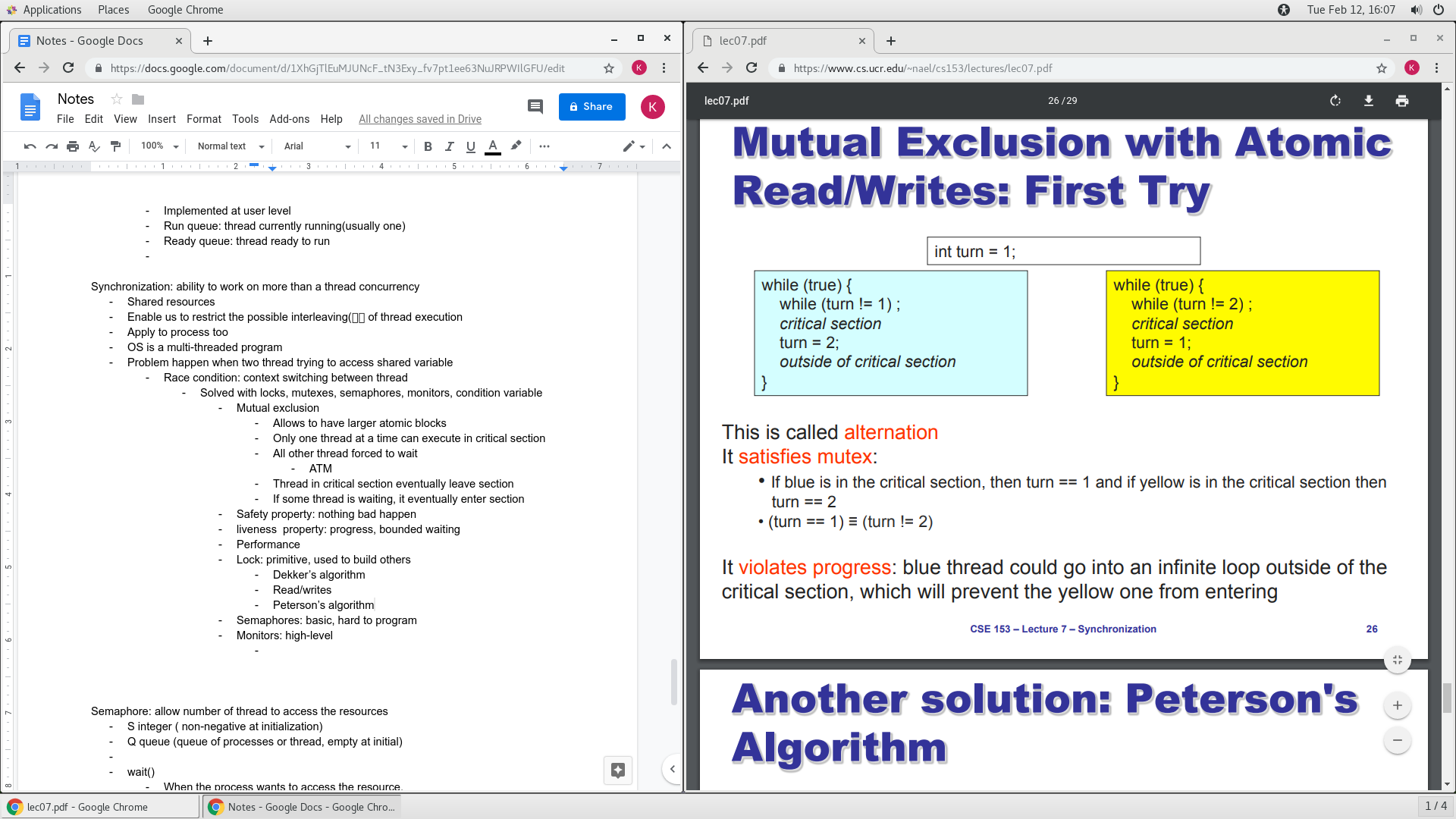
* Unit of scheduling
* Multi Threaded application
  + Share resources, access shared data structure
    - Web server
  + Concurrency does not require creating new thread
  + Improve program structure
  + handling concurrent event
  + Writing parallel program
  + Very useful on uniprocessor
* Kernel level thread
  + OS schedule all thread
  + Light weighted thread
* User-level thread
  + Implemented using user level library
  + Small and fast
  + Represent in (TCB) thread control block
  + Done via procedure call
  + 100x faster than kernel thread
  + No integrated with OS
* Thread interface
  + thread\_fork(procedure\_t)
    - Create new thread of control
    - thread\_create(), thread\_setstate()
  + thread\_stop()
    - Stop the calling thread
    - Thread\_block
  + thread\_start(thread\_t)
    - Start the given thread
  + thread\_yield()
    - Voluntarily give up the processor
    - Context\_switch
      * Saved current running thread
      * Restore context of next thread
      * Next\_thread become current thread
      * Return to caller as new thread
  + thread\_exit()
    - Terminate the calling thread
    - thread\_destroy
* Thread scheduler determines when thread runs
  + Use queue to keep track of what thread is doing
  + Implemented at user level
  + Run queue: thread currently running(usually one)
  + Ready queue: thread ready to run

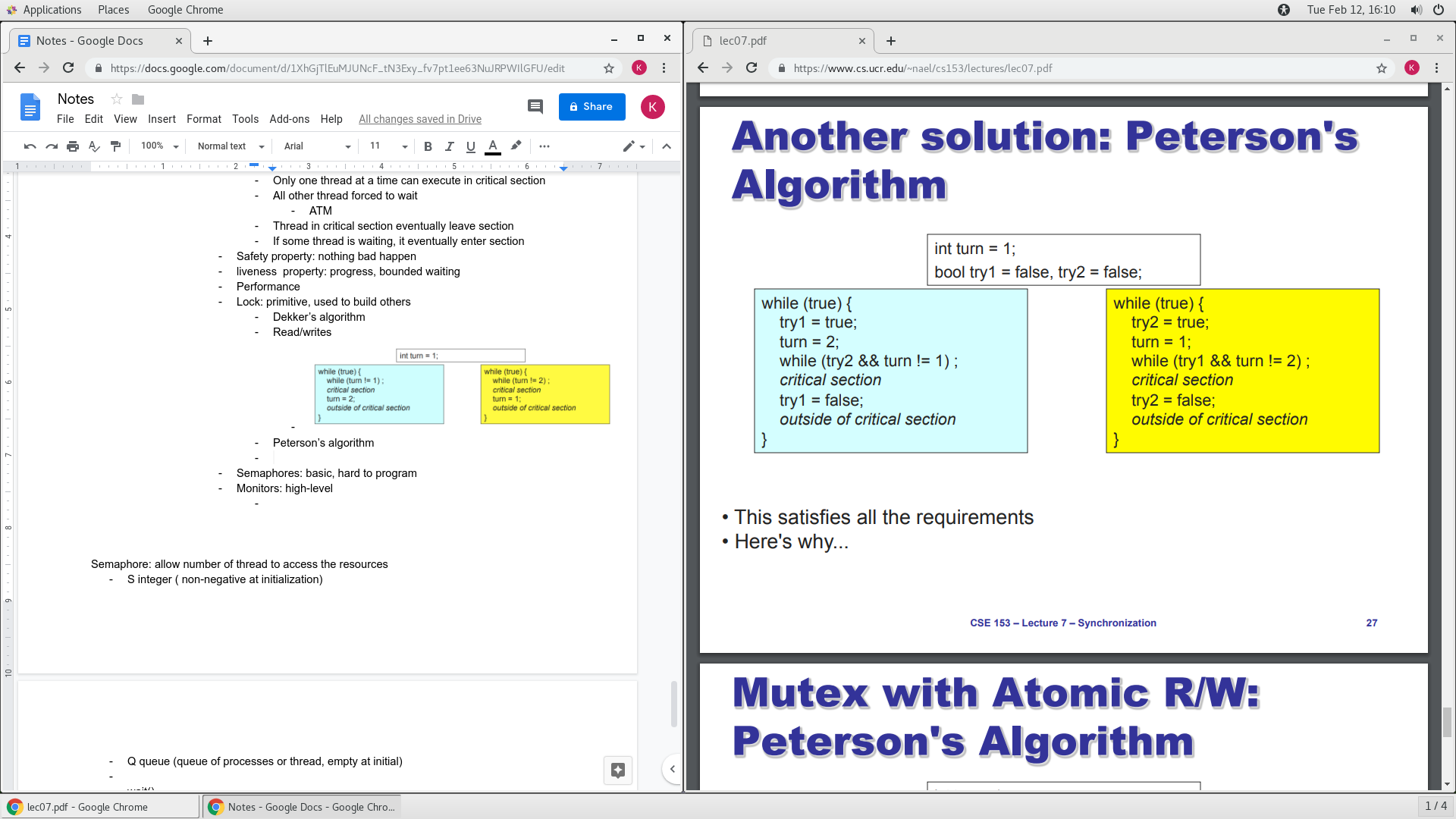
Synchronization: ability to work on more than a thread concurrency

* Shared resources
* Enable us to restrict the possible interleaving(交錯 of thread execution
* Apply to process too
* OS is a multi-threaded program
* Problem happen when two thread trying to access shared variable
  + Race condition: context switching between thread
    - Solved with locks, mutexes, semaphores, monitors, condition variable
      * Mutual exclusion
        + Allows to have larger atomic blocks
        + Only one thread at a time can execute in critical section
        + All other thread forced to wait

ATM

* + - * + Thread in critical section eventually leave section
        + If some thread is waiting, it eventually enter section
      * Safety property: nothing bad happen
      * liveness property: progress, bounded waiting
      * Performance
      * Lock: primitive, used to build others
        + Dekker’s algorithm
        + Read/writes



* + - * + Peterson’s algorithm
        + 
        + Lock doesn’t really work on software, need hardware

Software locks reordered by compiler/hardware

Spinlock

Thread waiting to acquire lock spin

Waste CPU cycle

Disable interrupts

Shouldn’t disable interrupt for long period of time

Can misdelay

* + - * Semaphores: basic, hard to program
        + Abstract data type
        + Block waiter, interrupt enabled within critical section

* + - * Monitors: high-level
        + object designed to be accessed from multiple threads. The member functions or methods of a monitor object will enforce mutual exclusion, so only one thread may be performing any action on the object at a given time.

Semaphore: allow number of thread to access the resources

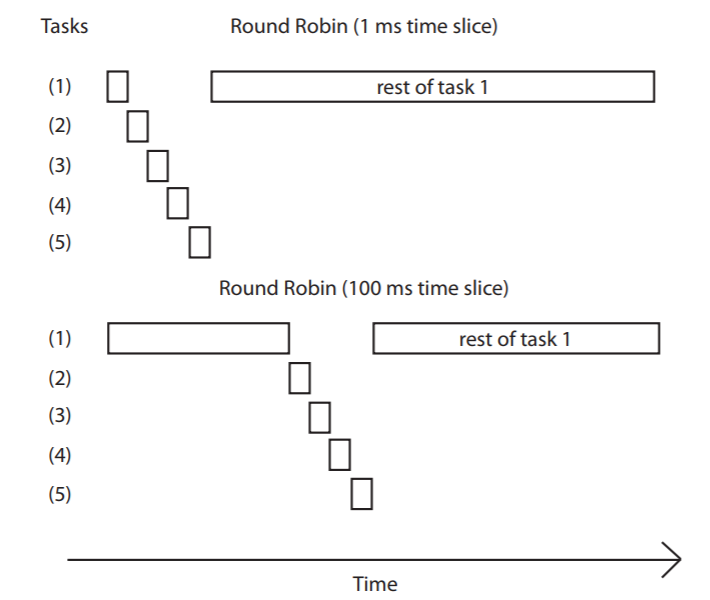
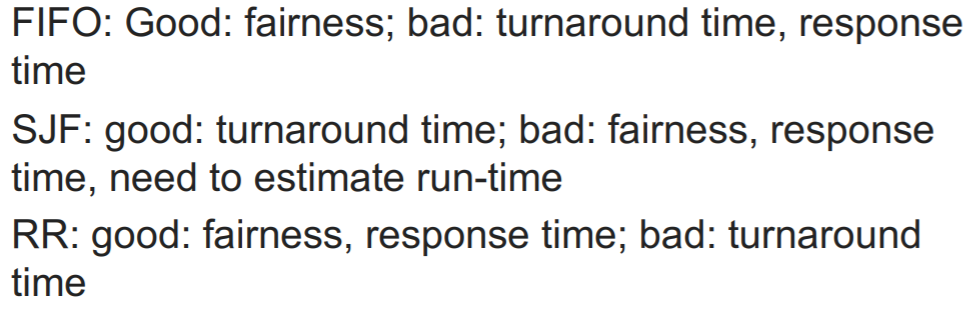
* S integer ( non-negative at initialization)
* Q queue (queue of processes or thread, empty at initial)
* wait()
  + When the process wants to access the resource.
  + Decrement S when called
    - If S is 0, then the process or thread is blocked and placed into Q
      * When in queue, these process is waiting for resources
* signal()
  + Is called when process done using resources
  + Increment S when called, if S is 0, next blocked process is dequeue from Q and become unblock
* Strong semaphore if queue is first in first out

Mutex: allow only one thread to access the resource

* Lock

Monitor: mutual exclusion

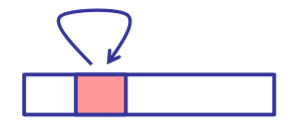
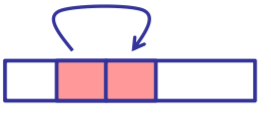
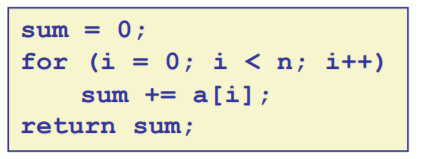
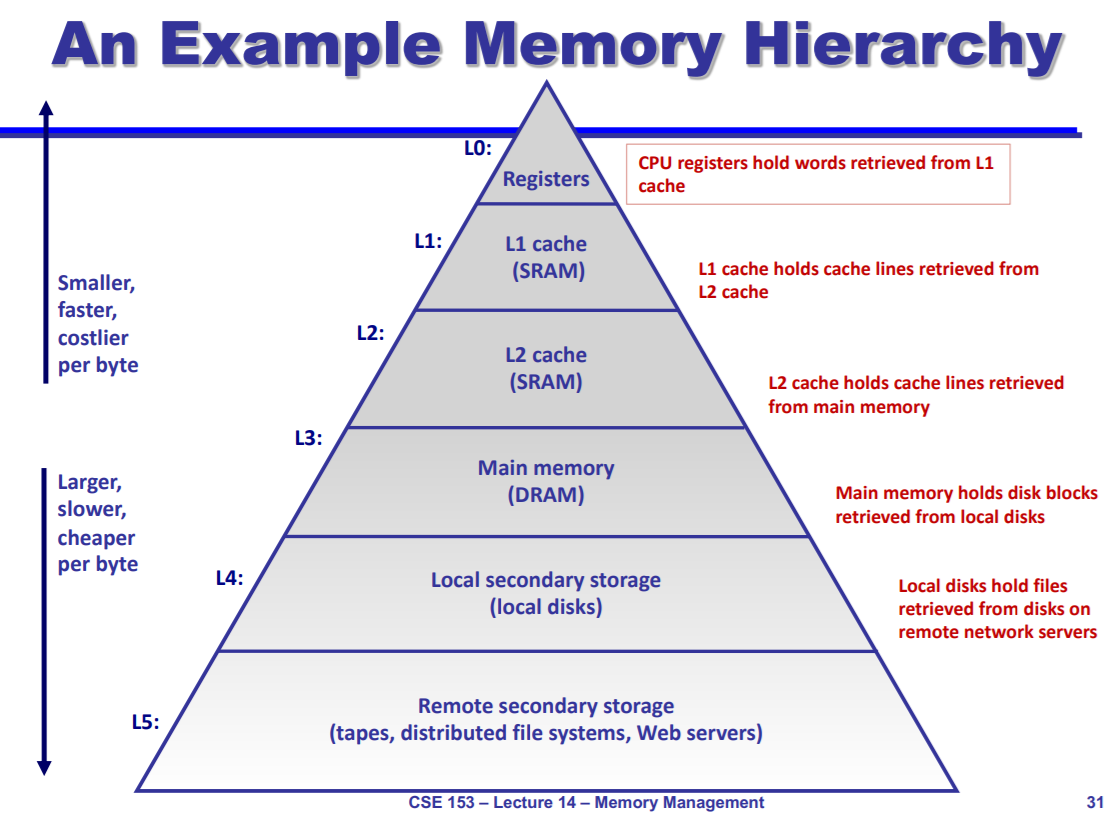
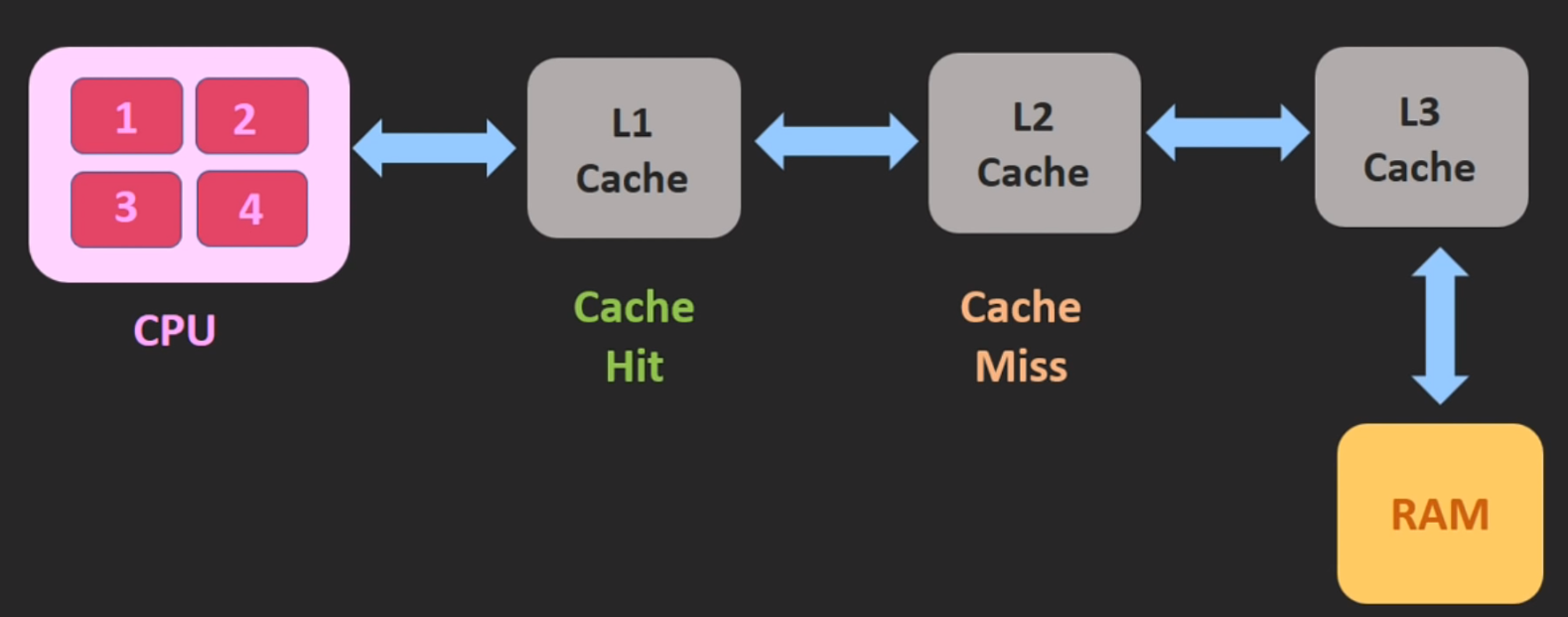
scheduling :

* Control multiprogramming level
  + Swapping
* Decide what job to run next
* Run when a job from running to waiting
* Interrupt occurs
* Job created or temrinated
* Turnaround time ( Tfinish - Tstart)
  + Turnaround time/ process length
* Avg waiting: avg time spend on wait queue
* Avg response time : avg time spent on ready queue
* Starvation: process prevented from making progress because other process have the resource it requires
  + High priority process always prevents a low priority process
  + Side effect of synchronization
* First in first out
  + Schedule task in the order they arrive
* Shortest job first(SJF)
  + Often called shortest remaining time first
* Round Robin
  + Each task gets resource for a fixed period of time
  + 
  + Many context switch can be costly
* Priority scheduling
  + Choose next job based on priority
  + Starvation problem happen: low priority wait forever
    - Age process
      * Increase priority as a function of waiting time
        + Decrease priority as a function of cpu consumption
* 
* Scheduling algorithm can be combined
  + Multiple queue
  + Different algorithm for each queue
  + Multiple feedback queues(MFQ)
    - No need to estate run time
    - Responsiveness
    - Low overhead
    - Starvation freedom
    - Some tasks are high/low priority
    - Fairness
    - High priority queue have short time slices

Deadlock:

* Mutual exclusion: at least one resource must be held in a non-sharable mode
* Hold and wait: must be one process holding one resource and waiting for another resource
* No Preemption: resource can not be preempted
* Circular wait: exist a set of process such that P1 wait for P2, P2 wait for P3
* Ensure at least one of the condition above does not exist can prevent deadlock
  + Mutual exclusion: make resources sharable
  + Hold and wait:process cannot hold one resource when requesting another
  + Preeemption: OS can preempt resource
  + Circular wait: impose an ordering on resources
* Described using resource allocation graph(RAG)
  + If graph has no cycle, no deadlock
  + If graph has cycle, deadlock may exist
* Banker’s algorithm
  + Assign credit limit to each customer(process)
  + Reject any request that can lead to dangerous state
  + Keep resource surplus
* Detect and recovery
  + One algorithm determine whether a deadlock has occured
    - Traverse resource graph look for cycles
      * If found, preempt resource
  + Another recover from the deadlock
    - Abort process
    - Preempt resources

Random-Access Memory:

* Basic storage unit is a cell
* Multiple ram chips forms a memory
* Volatile Memories: lose info without power
  + Static Ram(sram)
    - Each cells stores a bit with four or six transistor
    - Insensitive to electrical noise, radiation
    - Faster and expensive than DRAM
  + Dynamic Ram(DRAM)
    - Each cell stores a bit with a capacitor. One transistor is used for access
    - Value must be refreshed every 10-100ms
    - More sensitive to EMI than SRAM
    - Slower and cheaper
* Nonvolatile: keep info without power
  + Read-only memory(ROM): programmed during production
  + Programmable ROM(PROM): can be programmed once
  + Erasable PROM(EPROM): can be bulk erased(UV, X-ray)
  + Electrically erasable PROM(EEPROM): electronic erase
  + Flash Memory: EEPROM with partial erase capability
    - Wears out after 100,000 erasings
  + Phase change memories (PCMs): also wear out
  + Firmware programs stored in a ROM (BIOS,controllers for disk, network card
  + Solid state disk (replace rotating disk)
  + Caches in high end system
* Locality: program tend to use data with addresses near or equal to those they have used recently
  + Temporal locality
    - Recently referenced items are likely to be referenced again in the near future
    - 
  + Spatial locality
    - Items with nearby addresses tend to be referenced close together in time
    - 
  + 
    - Data locality
      * Temporal locality: sum, which is using over and over
      * Spatial locality: a[ ] which is succession with +1
    - Instruction locality
      * Temporal locality: cycle through loop repeatedly
      * Spatial locality: reference instruction in sequence
    - Memory Hierarchy: an approach for organizing memory and storage system
      * Locality: Higher level run more frequently
      * 
    - Cache: smaller, faster storage device that act as a staging area for a subset of data in larger, slower device
    - 
      * The lower level of memory hold data that higher memory can retrieve
        + Miss: If data is not found in lower level of cache

Cold miss: occurs when cache is empty

Conflict miss: level k cache is large enough,, but multiple data objects all map to the same level k block

Capacity miss: set of active cashe block is larger than the cache

* + - * + Hit: data is found in lower level of cache

CPU->find data in L1, if not found:miss-> L2, found, Hit

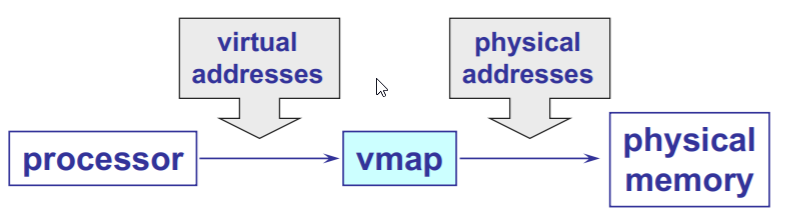
* + - * + If data not in cache, cache fetch data from memory

This might kick out data in block that has not been using for a while

Placement policy: determines where data in the block is pointed to

Replacement policy: determines which block get evicted

VIrtual Address: logical address

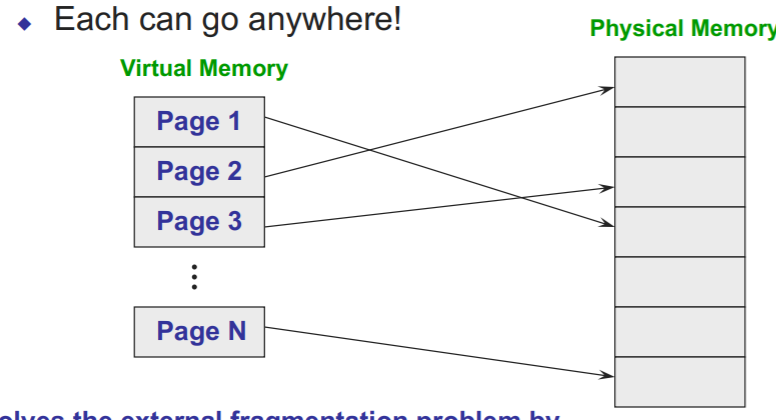
* Physical address is determined by OS
  + Translated by hardware into physical address
* 
* Requirement
  + Protection: restrict which address jobs can use
  + Fast translation: lookup need to be fast
  + Fast change: updating memory hardware on context switch
  + Partition
    - Size of each partition is the same
    - Hardware requirement: base register
    - Physical address = virtual address register
    - Easy to implement
    - Problems
      * Internal fragmentation
        + Memory in a partition not used by a process is not available to other processes
        + Partition size

One size does not fit all

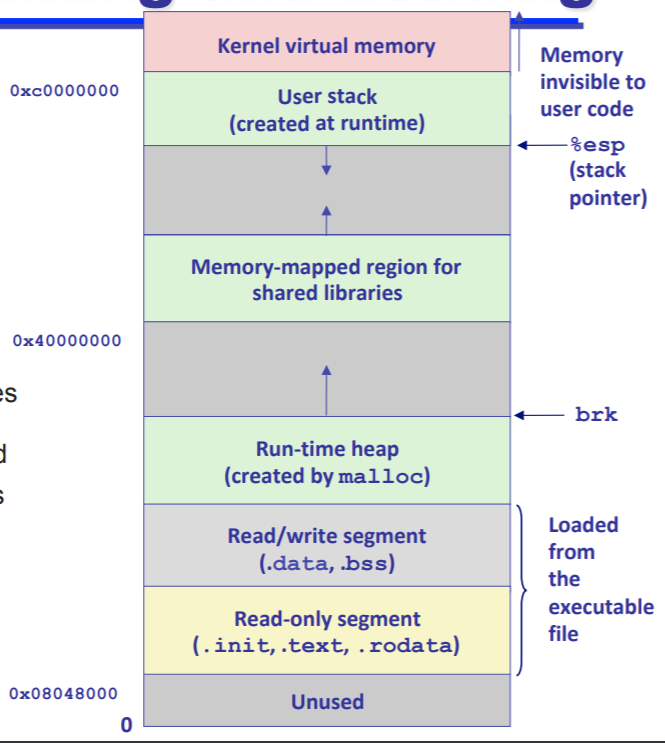
* + - * + Natural extension: Physical memory is broken up into variable sized partitions
      * Limit register base size can fix internal fragmentation
        + External fragmentation
* VA is an array of N contiguous bytes

Job loading and uploading produces empty hole scattered through memory

Paging: split virtual address space into multiple partitions

* Solve external fragmentation
* 
* Easy to allocate memory
* Don’t need a limit register
* Can have internal fragmentation
  + Process may not use memory in multiples of a page
* Use Copy on write(CoW) to deffer large copy
  + Create shared mapping of parent page in child virtual address space
    - Shared pages are protected as read-only in parent and child
* Page replacement policy: determine which page to remove when we need a victim

Memory stack

* 
* Before handling page fault, add protection for writing

Translation lookaside Buffer(TLB):

* Small hardware cache in MMU
* Miss still can happen
  + MMU load PTE from page table
  + Software managed TLB, OS intervenses at this point

Multi level page table, save more memory spaces

Belady’s algorithm

* Replace page that will not be used for the longest time in the future

Least recently used(LRU) :

* Too costly

Victim buffer: add a buffer