CSC 139 – Final Study Guide

Ch1 – Introduction:

- **OS Definition:** There is no universal definition for a OS, some say it is everything a vendor ships when you order an OS, others say it is the one program running at all times in your computer. OS is a resource allocator and control program.
- **OS Goals:** The goal of an OS is to execute programs and make problem solving simpler. It should make the computer more convenient to use and use HW in an efficient manner.
- **Kernel:** This is the one program that is always running on the computer. Other programs are either a system program (ships with OS) or an application program.
- **Symmetric Multiprocessing** is where all the processors are peers. All kernel routines can execute on different CPU's in parallel. In **Asymmetric Multiprocessing** the computer is structured using master/slave. The kernel runs on a particular processor, and CPU's can execute user programs and OS utilities. Both are **parallel systems** that give increase throughput, economy of scale, and reliability.
- **Interrupts:** Raised by external events, an interrupt informs the kernel to switch to another process and to perform some necessary interaction (I/O), once completed the CPU switches back to the running process. This enables the CPU to gracefully handle long-latency events. A modern OS is interrupt driven.
  - **Polling:** System determines interrupt by polling, which checks every so often for interrupts. This is proactive and wasteful of system resources
  - A **vectored interrupt system** lets the device inform the CPU of an interrupt, this is reactive and more efficient.
  - **Interrupt Vector:** Contains the address of the interrupt service routine to handle the type of interrupt received.
  - A **trap** is a software-generated interrupted caused either by an error or user request.
- **Dual Mode:** This form of operation allows the OS to protect itself from harm by using two modes (**user** & **kernel mode**) where some system operations are privileged and require kernel mode to execute. This can prevent resource hogging and infinite loops.

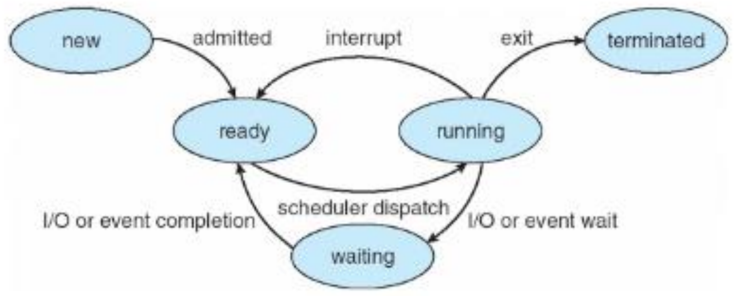
Ch2 – OS Structures:

- **System Calls:** Allow applications to access privileged mode instructions through services provided by the OS. Generally accessed via high level API's. These calls transfer control to the OS and raises the HW privilege level. System calls are the only way for a user program to access privileged instructions.
  - **Procedure Call:** Happens in user mode, merely changes the execution location of a program.
  - **System Call APIs:** Win32 (Windows), POSIX (POSIX-based systems, Unix, Linux, MacOS), and Java (JVM). Runtime libraries provide a system call interface that intercepts functions calls in the API and invoke the necessary system call in the OS. APIs encapsulate many of the low level and OS specifics.
  - **Parameter Passing:**
    - Pass variables directly to a register (Advantage: simple, Disadvantage: capacity is small).
    - Store in a block of memory and pass address of memory to a register (Advantage: no parameter limits, done by Linux and Solaris).
    - Place parameters on top of the program stack, parameters popped off by OS.

- **OS Layered Approach:** OS divided into layers, bottom layer is HW, top is user interface.
    - **Pros:** Easy to debug and maintain. Hides low level details.
    - **Cons:** Difficult to divide. Communication between layers introduces overhead.
- **Microkernel:** Move as much from the kernel as possible into the user space. Used by MacOS.
    - **Pros:** Easy to extend micro kernel. Easier to port OS. More reliable and secure.
    - **Cons:** Performance overhead of user to kernel communication.
- **Modular Kernel:** Loadable kernel modules that are dynamically loadable. Used by Solaris.
    - **Pros:** Uses OOP approach. Each component is separate and talks to each over their own interfaces. Modules loaded only as needed.
    - **Cons:** Performance overhead due to module communication and module loading.

Ch3 – Processes:

- **Process:** A program in execution. It is an active entity with associated resources
- **Program:** A passive entity. Has the ability to become a process if executed.
- **Process States:**



    - **New:** The process is being created (in this state for a small amount of time).
    - **Running:** The process is being executed on a CPU
    - **Waiting:** The process is waiting for some event to occur (I/O, etc.).
    - **Ready:** The process is ready to execute and waiting to be assigned to a CPU via the CPU scheduler.
    - **Terminated:** The process has finished executing (in this state for a small amount of time).
- **Process Memory Layout:** Information about a process is held in a OS data structure called a PCB that is within the OS's address space.
- **PCB (Process Control Block):** Tracks execution and the location of each process. Created by the OS when a process starts and held within the OS address space. Contains process state (running, waiting, etc.), program counter, CPU register contents, CPU scheduling info (priorities, etc.), memory allocation info, accounting info (CPU cycles consumed, clock time, etc.), and I/O status info (open files…).
- **CPU Bound Process:** A process that spends most of its time doing computations. Typically defined by a few long CPU bursts.
- **I/O Bound Process:** A process that spends more time doing I/O than computations. Typically defined by many short CPU bursts.
- **Context Switching:** Allows CPU's to save the state of a process (inside a PCB) and load another processes PCB. During each switch, it must perform a save operation for the current process, and a load operation for the process to be executed. The more complex the PCB (larger) the slower the switch. System does no useful work while performing a context switch.
- **Long-Term Scheduler:** AKA the Job scheduler. Determines which process(s) are brought into main memory.

- **Short-Term Scheduler:** AKA the CPU Scheduler. Decides which process(s) to execute on the CPU first.
- **Process Creation in Linux:** Each process is created as a child of a parent. A child starts with an exact buy separate copy of the parent's address space. In Linux, all process are children of SystemMD or Init.
    - **fork():** Spawns a child, returns 0 to the child, and the child's address to the parent. Returns -1 on failure.
    - **exec():** Loads a binary file (program) into memory. This replaces the caller's address space, executing a new program.
    - **wait():** Halts a parent's execution until child terminates.
    - **abort():** Called by the parent to forcibly terminate a child.
    - **exit():** Terminates the calling process, returns status data via wait().
- **Inter-Process Communication:** Allows a process to communicate with another process. These processes are called cooperating processes.
    - **Message Passing:** Involves the kernel to send messages to another process. Each time a message is sent or received the kernel is involved. Message size is either fixed or variable. Generally slower and more expensive than the shared memory approach.
        - **Direct Communication:** Messages sent between processes explicitly. Each process must name the receiving process to send, and the sending process to receive.
        - **Indirect Communication:** Messages are delivered to ports (mailboxes) that are unique. Allows multiple processes to communicate easily.
    - **Shared Memory:** Involves the kernel (only at the beginning) to set up an area of memory that can be accessed by the cooperating processes. Generally, more complex to setup and deal with, but is faster.

Ch4 – Threads:

- **Thread:** Allows for multiple sequences of execution within the same process, to a large degree independent from each other. Lightweight when compared to process creation. Can simplify code and increase efficiency. Kernels are generally multithreaded.
    - **Share:** Address space, open files, Global variables, and Accounting Information.
    - **Unique:** Program Counter, Registers, Stack, and State.
    - **States:** Ready, Running, Blocked
    - **Pros:** Responsiveness, Resource Sharing, Economy, Scalability.
- **Kernel Threads:** A thread that the OS knows about. A lightweight process (relatively fast context switching). These threads are supported and managed by the kernel. Kernel manages and schedule threads just like processes (uses the same algorithms).
- **User Threads:** Managed by user-level thread libraries. Kernel is not aware of these threads, and only knows about the process that contains these threads. Therefore, the OS can only schedule the process to run, not the individual threads.
    - **Advantages:** No Context Switch involved. Scheduling is defined by the program (can be error prone). Cheaper than kernel threads (much faster).
    - **Disadvantages:** No true parallelism, OS scheduler could make poor decisions due to the hidden existence.
- **Many-To-One:** Many user level threads are mapped to a single kernel thread. A single blocking thread causes all threads to block. Threads cannot run in parallel as only one may be in the kernel at a time. Not generally used in Modern OS's.

- **Many-To-Many:** Many user level threads are mapped to many kernel threads (generally not one to one). Allows OS to create enough kernel threads to service the user threads.
- **One-To-One:** One user thread is mapped to a single kernel thread. This is more expensive but eliminates any issues due to blocking threads. Most popular in modern OS's.
- **Thread Libraries:** An API thar provides the programmer the tools for creating and managing threads. Library is entirely in the user space, with the Kernel-Level-Library supported by the OS.
  - **Pthreads:**
    - Can be either User-Level or Kernel-Level.
    - A POSIX standard for API thread creation with synchronization.
    - API specifies behavior of the thread library. Implementation is OS dependent.
    - Common in Unix OS (Linux, Solaris, MacOS).
  - **Java Threads:**
    - Managed by the JVM
    - Typically implemented using a thread model provided by the OS
- **Implicit Threading:** Creation and management of threads done by compilers and run-time libraries rather than the programmers. Generally, simplifies the program and increases correctness.
  - **Thread Pools:** Create a number of threads where they await work.
    - **Advantages:** Slightly faster to service a request as a thread has already been created. Allows the number of threads to be bounded. Separates the mechanics of creating a thread from the running task.
  - **OpenMP:** A set of compiler directives for C, C++, and FORTRAN that provide support for parallel programming. Identifies parallel regions using blocks. Creates as many threads as there are cores.
    - **#pragma omp parallel:** Defines a parallel block.
  - **Grand Central Dispatch:** Identifies parallel regions as blocks. Parallel blocks placed in a concurrent queue where multiple processes can be removed and ran at the same time. Non-parallel processes are placed in a serial queue where processes can only remove one at a time.
- **Threading Issues:**
  - **fork():** Does fork duplicate all the processes threads? Depends on OS implantation. Some Unix versions have two versions
  - **exec():** Works normally. Replaces the running process (including all threads).

Ch5 – CPU Scheduling:

- **Scheduler Run Conditions:** Under the state transition model presented earlier, a CPU scheduler can (but not always) run when one of the following happens:
  - When a process switches from the running state to the waiting state (Both P and Non-P).
  - When a process switches from the running state to the ready state (Only P)
  - When a process switches from the waiting state to the ready state (Only P)
  - When a process terminates (Both P and Non-P)
- **CPU Scheduler:** Selects a process from the ready queue to execute on the CPU. The algorithm used for determining selection may have a tremendous impact on system performance.
- **Dispatcher:** Gives control of the CPU to the process selected by the CPU scheduler. It performs the context switch, the switch to user mode, and jumping to the proper location in the user program to restart the program. **Dispatch Latency** is the time it takes for the dispatcher to stop and run another process.

- Under **Non-Preemptive Scheduling** each running process keeps the CPU until it completes or switches to a waiting or blocked state. Under **Preemptive Scheduling** a running process may be forced to leave the CPU even if it is neither blocked nor completed. This type of scheduler generally runs whenever the ready queue changes and/or the current processes time quantum expires.
- **Scheduling Criteria:**
  - **CPU Utilization:** Keep the CPU as busy as possible (Maximize).
  - **Throughput:** Number of processes that complete execution per time unit (Maximize).
  - **Turnaround Time:** Amount of time to execute a particular process from start to finish (Minimize).
  - **Waiting Time:** Amount of time a process has been waiting in the ready queue (Minimize).
  - **Response Time:** Amount of time it takes a request to first be serviced by a CPU (Minimize).
  - **Meeting the Deadline:** Only in Realtime systems.
- **First Come First Serve (FCFS) Scheduling:**
  - First process to arrive, get assigned to the CPU.
  - Implemented with a simple FIFO queue.
  - Performance is highly dependent on the process arrival order.
  - **Advantages:** Simple.
  - **Disadvantages:** Convoy effect, short process behind a long process. May lead to poor overlap of CPU bound and I/O bound processes. Leaving I/O devices idle.
  - Non-Preemptive
- **Shortest Job First (SJF) Scheduling:**
  - With each process the length of its next CPU burst is associated. The process with the shortest next burst time runs first.
  - Optimal given that the processes arrive at the same time.
  - Difficult to implement since it is almost impossible to know the length of the next CPU burst.
  - Non-Preemptive
- **Shortest Remaining Time First (SRTF) Scheduling:**
  - Preemptive version of SJF, scheduler runs each time a process enters the ready queue.
  - Optimal given the processes arrive at different times.
  - Difficult to implement due to the same issue with SJF.
- **Round Robin (RR) Scheduling:**
  - Each process runs for a single time quantum (time unit). When this runs out, the process is preempted even if execution has not finished.
  - If N processes are in the ready queue, with a time quantum = q. Each process waits for **1/N CPU in chunks of q**. No process waits for longer than **(N – 1)q** time.
  - Always Preemptive
  - **Time Quantum:**
    - **Large:** FIFO performance
    - **Small:** Large overhead due to the amount of context switches.
    - **Rule of Thumb:** TQ typically larger than 80% of processes CPU run time (allow I/O bound process to run in a single TQ) to prevent the large overhead of context switches.

- **Priority Scheduling:**
  - Each process has an associated priority (typically an integer). The process with the highest priority always runs first.
  - Can be Preemptive or Non-Preemptive depending on implementation.
  - **Problems:**
    - **Starvation:** A low priority process may never execution
    - **Aging**: A processes priority increases as time progresses. Solves starvation.
- **Multilevel Queue:**
  - Ready queue is partitioned into separate queues with different priorities and scheduling algorithms.
  - Processes are permanently assigned to a queue
  - **Scheduling Between Queues:**
    - **Fixed Priority:** A higher priority queue must be empty before a lower priority queue can execute its processes.
    - **Time Slice:** Each queue gets a time slice where it can run its processes.
  - Suffers from starvation
- **Multilevel Feedback Queue:**
  - Processes can move between the various queues (can implement aging).
  - **Defined by the following:**
    - Number of queues. Scheduling algorithms for each queue. Method used to determine when to update/demote a process. Method to determine which queue a process will enter when it needs service.
- **Linux O(1) Scheduler:** Used in Linux up to version 2.5. Called a constant time scheduler
  - Preemptive & Priority based
  - Realtime priorities range from 0-99. Default user priorities range from 100-140 (-20 to +19 depending on niceness values).
  - A higher priority process gets a bigger time quantum and vice versa.
  - The longer a process sleeps the higher its priority and vice versa.
  - **Implementation:**
    - Two priority arrays (active/expired). Tasks indexed by priority in the array. When no more processes are active, the arrays are exchanged.
    - O(1) to find a process to execute. O(1) to add a process.
  - **Issues:** Not fair, poor performance for interactive processes.
- **Linux Completely Fair Scheduler (CFS):**
  - Uses a red-black tree ordered by virtual runtime (vruntime) with the leftmost node always holding the process with the lowest vruntime.
  - vruntime progression depends on priority and niceness. Progresses faster on a lower priority process and vice versa.
  - Preemptive, the process with the lowest vruntime always runs first
  - Requires only a single tree to capture all the processes.
  - Worst case O(log N) to add a process, O(1) to select due to a tail pointer.

Chapter 6 & 7 – Process Synchronization:

- A **race condition** occurs when multiple processes are writing or reading shared data and the result depends on the execution order of the processes
- A **critical section** is a section of code in which processes access or modify shared data.
- **Test-And-Set (TAS)** HW instructions test and modifies the content of a memory word atomically (in an arbitrary order). Therefore, only a single process can modify a variable at a time.
- **Mutex locks** are a simple OS tool that allow process to *acquire()* or *release()* a lock on a critical section. Call must be atomic.
  - **Advantage:** Simple | **Disadvantage:** Spin locks with busy waiting
- **Semaphores** are a non-negative integer that have two valid operations
  - **wait():** If semaphore > 0 then semaphore **-** 1. Else block this process
  - **signal():** If there is a blocked process wake it up, else s += 1;
  - **Binary sems** can only be 0 or 1. **Counting sems** can be any non-negative integer.
  - **OS service** generally implemented through disabling interrupts for a short time.
  - **Advantages:** Avoids busy waiting! | **Disadvantages:** Error prone & difficult to use.
- **Starvation** occurs when a process is infinitely delayed in favor of other processes. Generally caused by a bias in system scheduling / resource or waiting techniques.
- A **deadlock** occurs when a cycle exists. A locked resource is wanted by a process that is held by another process that wants the resource the original process holds.
- **Classical IPC Problems & Solutions:**
  - **Bounded Buffer:** A producer and consumer share a bounded circular buffer. Producer puts items in, consumer removes items. How do you prevent race conditions if these two processes run in parallel?
    - **Solution:** Three semaphores. Two are counting and keep track of how many items are in the buffer and how many free spots there are, respectively. One is binary to provide mutual exclusion for interacting with the buffer.
  - **Reader-Writer:** Let multiple readers read, but only one writer right at a time.
    - **Solution 1:** A writer can only enter when no readers are reading. Therefore, it must wait until no readers are active. Requires 3 semaphores or mutex locks.
    - **Solution 2:** If a writer is waiting, prevent readers from entering. Writer proceeds when no readers remain in the critical section.
  - **Dining Philosopher:** Five philosophers sitting at a round table, randomly choose to eat. 5 chopsticks, 2 chopsticks required to eat.
    - **Solution:** Label each philosopher ODD or EVEN. Had them alternate the chopstick they pick up first. One semaphore per chopstick. If a chopstick is being used wait, Else signal and pick it up.
- **Monitors** are a language construct that include synchronization. Only 1 process can enter the monitor at a time (OS implementation could be a semaphore or condition variables).
- **Condition Variables:** A variable where only two operations are allowed
  - **cond.wait():** Invoking process is suspended until **cond.signal()**
  - **cond.signal():** Wakes up a process that invoked **cond.wait().** Otherwise it does nothing.
  - **What occurs if a process invokes signal() and a process is already suspended in wait?**
    - **Signal & Wait:** Signaling process suspended, suspended process executed
    - **Signal & Continue:** Signaling process allowed to continue, suspended process must wait.

Chapter 8 – Deadlocks

- **Deadlock Characterization (deadlock can occur if all 4 hold true):**
  - **Mutual Exclusion:** 1 process can use 1 resource at a time.
  - **Hold & Wait:** A process holding a resource is waiting to acquire another locked resource.
  - **No preemption:** A resource can only be released voluntarily by the process holding it.
  - **Circular Wait:** There exists a set of processes such that P0 is waiting for a resource held by P1 which is for a resource held by PN -1 and PN is waiting for a resource held by P0.
- **Resource Allocation Graph Basic Facts**
  - If a graph contains **no cycles…** No deadlock.
  - If a graph **contains a cycle…**
    - If only 1 instance per resource type the system is deadlocked
    - If several instances per resource type the system has the possibility of deadlock
- **Handling Deadlocks** (Three approaches)
  - **Deadlock Prevention:** Ensure a deadlock will never occur (invalidate 1 of 4 deadlock characteristics)
  - **Deadlock Avoidance:** Never allow the system to enter an unsafe state.
  - **Ignore it:** Done by most OS's as the above operations are extremely expensive.
- **Deadlock Prevention** – Prevent deadlocks by removing the possibility for them to occur
  - **Mutual Exclusion**: Not required for sharable resources. There is no possibility of deadlock. However, must hold for non-shareable resources…
    - **Pros**: Solves deadlocks with little expense | **Cons:** Only can invalidate for sharable resources
  - **Hold & Wait:** Whenever a process requests a resource, it can hold none. Allocate all required resources in one step.
    - **Pros:** Solves deadlocks | **Cons:** Low resource utilization with possibility of starvation
  - **Preemption:** If a process that is holding resources requests more that are held by other processes… Release all currently held resources and put the process to sleep. Only wake when all previous resources held and requested resources are available.
    - **Pros:** Solves deadlocks | **Cons:** Possibility of starvation
  - **Circular Wait:** Impose a total ordering of all resources (most common method). Require each process request a resource in increasing order
    - **Pros:** Solves deadlocks with little overhead | **Cons:** Resources allocated on FCFS basis, possibility of starvation and can be extremely complex to implement.
- **Deadlock Avoidance**
  - **Safe State:** No possibility of deadlock (there exists a safe sequence)
  - **Unsafe State:** Deadlock possible
  - **Safe Sequence:** A sequence of **ALL** processes running that all can complete with the current state of the system.
  - **Avoidance:** Never allow the system to enter an unsafe state
- **Banker's Algorithm**
  - Proposed by Dijkstra. Can be used to determine if the system is currently in a safe state.
    - **Pros:** Allows deadlocks to be avoided before they occur | **Cons:** Extremely expensive to run during every resource request as the algorithm is **$O(m \times N^2)$**
- **Deadlock detection & recovery:**

- Depending on when the detection algorithm is invoked it may be extremely hard to figure out which process caused the deadlock.
- **Could:** Abort all deadlocked processes or abort a single process at a time until deadlock cycle is eliminated, victim order (priority, execution time, time remaining, resources held, interactive or batch?)
- **General rules:** Select a victim with minimal cost, return to some safe state and restart the process for that state, ensure no process is always picked as the victim (#rollbacks variable).

Chapter 9 – Memory Management

- **Logical Address:** AKA virtual address, generated by a CPU.
    - **Logical Space:** Set of all logical addresses generated by a program
- **Physical Address:** Actual address on the memory unit.
    - **Physical Space:** Set of all physical addresses generated by a program
- **Binding Addresses:**
    - **Compile Time:** Compiler generates the exact physical location in memory starting from some fixed position. The OS does nothing.
    - **Load Time:** The compiler generates an address; at load time the OS determines the process's starting location. Once loaded the process cannot be moved.
    - **Execution Time:** The compiler generates an address; the OS can place it anywhere in memory.
- **Static Relocation:** OS adjusts process addresses to assigned location in memory. Once a process begins execution it cannot be moved since the actual program was changed.
- **Dynamic Relocation:** Uses a base and limit register for each process to define a "space" where the process can run. The space between these two registers is its address space. Each process holds unique values in its registers.
    - **Advantages:** Process can be moved during execution; Process can grow over time. Simple and fast (only 2 HW registers).
    - **Disadvantages:** Slow down due to ADD on every memory reference, can't share memory, limited by physical memory size, complicates memory management.
- **Segmentation Architecture:**
    - A program is split into logical units (of varying size) that are called **segments**.
    - Each **segment** has a unique **base** and **limit** register.
    - Addresses stored in a **segment table** in a two tuple in form **<segment#, offset>**
    - Segment table stored in memory, start defined **by Segment-Table-Base-Register** (location of table) and **Segment-Table-Length-Register** (number of segments in table).
    - **Pros:** Allows for dynamic, non-contiguous allocation of memory | **Cons:** Suffers from external fragmentation.
- **Paging Architecture:**
    - **Logical memory** is contiguous, and divided into fixed size **pages**
    - **Physical memory** is divided into blocks of the same size called **frames.**
    - Any page can be mapped to any frame.
    - **Pros:** Avoids external fragmentation and allows for dynamic, non-contiguous memory allocation | **Cons:** Suffers from internal fragmentation.

- **Page Tables:**
    - A **per-process** data structure used to keep track of virtual page to physical frame mapping
    - **Address translation:**
        - **Page Number (p):** Index in a page table which contains the base address of each page in physical memory.
        - **Page offset (d):** Combined with the base address to define the physical memory address sent to the memory unit.
        - **Page table entry:** Each entry holds the physical translation and other info. The **Page-Table-Base-Register** points to the page table, the **Page-Table-Length-Register** indicates the size of the page table.
        - **Page Table Entries:** $2^{\text{logical address size}}$ / page size (ex: 4KB = 2^{12})
        - **Page Table Total Size:** #entries * (entry Size)
        - **Offset Length:** Log_2(page size) = #offset bits
- **Contiguous Memory Allocation:**
    - **First-Fit:** Allocate process in the first available hole big enough to fit the process (begin search at first hold or where previous first-fit ended).
    - **Best-Fit:** Allocate the smallest hole that is big enough to hold the process. OS must search entire list, or store list in a sorted manner.
    - **Worst-Fit:** Allocate the largest hole, OS must search entire list or keep it sorted.
    - **Internal Fragmentation:** Exists when memory in an internal partition is wasted.
    - **External Fragmentation:** Exists when there is enough memory to fit a process, but not contagiously. Generally, for every 2N blocks, N are lost due to fragmentation (1/3).
- **Translation Look-Aside Buffer (TLB)**
    - Used to speed up memory translation, it is a fast-look up HW cache located on the CPU
    - **TLB Hit**: TLB holds desired entry and physical frame number can be extracted from the TLB.
    - **TLB Miss:** TLB does not hold the address, must look up inside the page table. Update TLB with new translation.
    - **Effective Access Time:** Hit ratio * TLB access time + miss ration * (TLB Access time + memory access time)
- **Valid/Invalid Bit:** Attached to each page table entry, valid indicates page is in the process' logical address space and is legal, invalid indicates the page is not in the logical address space and invalid. Also used to determine if a reference page is stored in memory or not.
- **Multilevel Page Tables:** Break up logical address space into multiple page tables.
    - **Two Level:** Outer page table entries indicate the location of the second-level page table which in turn holds the physical address translation
- **Hashed Page Tables:**
    - The virtual page number is hashed into a page table, collision are chained together at collision point.
    - Advantage: Allows the size of the hashed table to be bounded, Worst case performance O(N)
- **Inverted Page Tables:**
    - One single page table with a single entry for each real page of memory.
    - Entries consist of physical address translation info and the PID of the process that owns it.
    - **Advantages**: Size of page table bounded to memory size |**Cons**: Slow look-up and difficult to share memory between processes.

Chapter 10 – Virtual Memory

- **Virtual Memory:** Separates user logical memory from physical address.
  - **Advantages:** Logical address space can be larger than physical address space. Only parts of the program need be in memory, address spaces easily shared, efficient process creation, more concurrent programs, less I/O need to load and swap processes.
- **Page Fault:** Occurs when a process tries to reference a page table entry with an invalid bit set to "invalid". Results in a page fault that the OS handles. If the address accessed is not part of the process' logical address space the OS returns an error and aborts the process. Else, the reference page must be out of memory, as a result it is loaded into memory and the process restarted at the instruction that caused the page fault.
  - **Effective Access Time:** page fault rate * memory access time + (page fault rate * page fault service time)
- **Demand Paging:** OS loads the page the first time it is referenced. It may remove a page to free up space. Process must give up the CPU while the page is loaded. Page faults occur…
- **Pre-Paging:** OS guesses in advance which pages will be needed and pre-loads them into memory. Allows more overlap of CPU and I/O if guess is correct, else a page fault occurs. Difficult to get right with code branching…
- **Page Replacement Algorithms:**
  - **FIFO:** Throw out the oldest page.
    - **Pros**: Simple to implement | **Cons**: Can easily throw out a page that is being accessed frequently.
  - **MIN (Optimal):** Throw out the page that will not be accessed for the longest time.
    - **Pros:** Always gives the optimal result | Cons: We can't see the future, impossible to implement.
  - **LRU:** Approx. of Optimal. Throw out the page that has not been used for the longest time.
    - **Pros:** Generally better than FIFO and is frequently used | **Cons:** Harder to implement than FIFO… Requires some form of HW support.
- **Kernel Memory Allocation:**
  - **Buddy Allocator:** Allocate memory from a fixed-size segment of physically contiguous pages. Memory allocated using the power of 2.
    - Satisfies request in powers of 2, if request is smaller than a power of 2 it is rounded up. When a smaller allocation is needed than available, the memory is split into chunks by dividing it into half until the memory is the right size. These chunks become buddies.
    - Empty buddies are merged into larger chunks…
    - **Advantage:** Allows unused memory to quickly coalesce into larger chunks
    - **Disadvantage:** Suffer from both external and internal fragmentation.
  - **Slab Allocator:** Create caches for each unique kernel data structure. Each cache is allocated one 1 or more slabs. When a cache is created all items are marked as free, if used marked as used.
    - If a slab is full of used objects, next object is allocated on an empty slab. If not, slabs are available… A new slap is allocated.
    - **Advantages:** No fragmentation, fast memory request satisfaction.

Chapter 11 – Mass Storage Structure

- **Disk data organization**: A disk is generally addressed as a large 1d array of logical blocks. Where each block is the smallest unit of transfer supported. These blocks are mapped into sectors, on tracks. Sector 0 is the first sector on the outermost track. Mapping proceeds in logical order on that track.
  - **Sector:** A logical block on the disk mapped onto a single track. Generally the smallest unit of transfer supported
  - **Track:** A circular lane where sectors are mapped. Outer tracks are larger due to a greater radius. This leads to the problem of having more blocks on outer tracks with 2 possible solutions
    - **Solution 1:** Reduce bit density per track for outer layers, results in a constant linear velocity, typically done on an HDD.
    - **Solution 2:** Vary the rotational speed basked on what track is being read. Results in a constant angular velocity, typically done on CD and DvD's.
  - **Cylinder:** The set of all the tracks at the same location on the entire disk stack (or platter).
- **Access time** on a HDD depends on the latency of the disk. This can be calculated using the following equation: $L_{I/0} = L_{seek} + L_{rotate} + L_{transfer}$
  - **Seek time:** Generally between 3-9ms, will probably be given to you in a problem
  - **Rotational Latency:** Generally given in RPM (rotation per minute). Multiply by *1/60,000* to get RPMS (rotation per MS). Divide this into 1 to get MS per rotation and divide by 2 to get the average.
  - **Data Transfer:** Generally relatively fast. Generally given in MB/S. Multiply by *1/1,000* to get MB/MS.
- Disk scheduling algorithms are used to minimize seek time as that is generally the only thing we can minimize through the OS. We do this by saving distance
  - **FIFO:** Handle request in-order, as they arrive
    - Pros: Simple
    - Cons: Generally, results in large head movement
  - **SSTF:** Go to the request that is closest to the current location of the head. Optimal in terms of distance but can result in starvation.
    - Pros: Optimal distance
    - Cons: Not fair, can result in starvation
  - **SCAN:** Disk head moves toward the other end and services requests along its way. Once the end is reached, it turns out and travels back servicing more requests.
    - Pros: Generally fair, eliminates starvation
    - Cons: If requests are uniformly dense, with the largest density at other end of disk, those will wait the longest.
  - **CSCAN:** Same as scan but returns to start and performs requests going down in the same direction
    - Pros: Avoids idle time introduced in SCAN
    - Cons: A giant leap is required, which takes time.
  - **C-LOOK:** Can be added to SCAN or CSCAN, means that the disk head will only go as far as the last request before turning around.
    - Pros: Reduces unnecessary movement
    - Cons: Still subject to the same issues as SCAN and CSCAN

Chapter 13-14 – File Systems

- **File structures:** A file system is an abstraction of the concept of storage, sectors, and tracks. The OS creates this abstraction by relating a collection of disk blocks (HW reality) to a bunch of named bytes (User view).
    - **File:** A logical collection of data that is name. Either contains a program or data. Has 3 names.
        - **inode:** The low level name, unique for all files at a given time. May be reused after deletes.
        - **Path:** Human readable name
        - **File Descriptor:** Holds the runtime state of the file
- Common **file operations** include create, delete, open, close, read, write, and seek. Common naming operations are hardlink, softlink, rename, set/get.
    - **Create():** Allocates disk space (if the space exists and it has permission to do so), creates a file descriptor (name, location on disk, all file attributes), Adds the file to the directory that contains that file, Sets optional file attributes.
        - int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC, S_IRUSR | S_IWUSR);
            - **O_CREAT:** Create if it doesn't already exists
            - **O_WRONLY:** Write only
            - **O_TRUNC:** If it already exists, truncate it to 0, meaning remake it or empty it
    - **Delete():** Finds the directory, frees the disk blocks used by the file, removes the file descriptor from the directory. Some behavior depends on hard links. Generally, not physically erased, just marks the sectors as deleted so overwrites can occur.
    - **Read():** Reads the file form the current position into a buffer (generally in memory)
    - **Write():** Similar to read, but writes the contents of a buffer to a file
    - **Seek():** Updates the location of the file pointer
- **Random Access** occurs when you are skipping around inside a file reading various locations (typically bad performance on an HDD), **sequential access** occurs when you read a file in order (as it appears on disk). The performance for this is fast (even on HDD's).
- The OS maintains two **file data structures:**
    - **Open file table:** Includes information on all open files system wide. Such as open count (how many processes have the file open), file attributes (ownership, protection, info, access times…), location on disk, and pointers to location of file in memory.
    - **Per process file table:** Maintained for individual processes with open files. For each file holds, a pointer entry to the current position in the file (offset), a pointer to the entry in the open file table, mode in which the file is open (R, W, RW), and pointer(s) to the file buffer.
- Directories are a special type of file that contains a list of tuples for files stored under them <high lvl name, low lvl name>. They support a few special operations such as traverse, and search.
    - **Single level directory:** OS maintains a single directory for all users.
        - Pros: Great for small disks, simple
        - Cons: Naming has to be unique across all users, single group
    - **Two level directory:** OS maintains a separate directory for each user
        - Pros: Path name, same file name for different users (path is unique), efficient searching.
        - Cons: No grouping capability, file sharing is tough.
    - **Multilevel (Tree) Directory:** Every node can have 0 or more children but will always have a single point

        - Pros: Directories in directories, naming is easy
        - Cons: Does not support sharing between users
    - **Acyclic (No cycles) Graph:** Same file can have multiple paths and therefore multiple parents
        - Pros: Enables sharing though hard and soft links
            - **Hard links:** Adds a second connection the file if, OS maintains a reference count. It will only delete the file when all references are gone.
                - Created with (LN cmd)
            - **Soft links:** Makes a symbolic pointer from 1 file to another. Deleting the file it is symbolically linked to will create an invalid pointer location.
                - Created with (LN-S cmd)
        - Cons: Complex
- **Directory Implementation:**
    - **Linear List** of file names with pointers to data blocks.
        - Pros: Simple
        - Cons: Time consuming to search (linear), could keep it ordered alphabetically via linked list or B+ tree.
    - **Hash Table**, a linear list with a hash data structure
        - Pros: Decreases directory search time
        - Cons: Collisions can occur reducing performance. Only good if entries are fixed size or a chain overflow method is used.
- File block allocation can vary. The OS must maintain an ordered list of free disks blocks
    - **Contiguous Allocation:** OS allocates a contiguous chunk of free blocks when it creates a file.
        - Pros: Only need to store the start location, and size of the file in the descriptor. Simple, supports sequential access and reduces the number of seeks
        - Cons: Changing file sizes generally require entire file to be rewritten, suffers from external fragmentation, need for compaction.
    - **Linked Allocation:** Keeps a list of free sectors and blocks, in each file descriptor store a pointer to the first sector/block, in each sector, keep a pointer to the next sector.
        - Pros: No fragmentation, file size is easy to change, non-contiguous scheme.
        - Cons: Only sequential access is supported, number of seeks are large as file is scattered across the disk
    - **Indexed Allocation:** OS keeps an array of block pointers for each file, OS allocates an array to hold the pointers to all blocks when it creates the file, OS files the pointer table as it allocates blocks to that file.
        - Pros: Not much wasted space, supports both sequential and random access
        - Cons: Wasted space in file descriptors, lots of seeks because data is scattered, sets a max file size
    - **Multilevel Indexed Allocation:** Each file descriptor contains pointer that point to more pointers allocated in a different block that can be allocated on demand.
        - Pros: Supports both large and small files, file can grow easily
        - Cons: Wastes some space in file descriptors, lots of seeks because data is scattered