CSC 139 – Quiz 2 Study Guide

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.

Ch6 – Synchronization:

- A **race condition** occurs when multiple processes are writing or reading shared data and the final result depends on the execution order of the processes.
- A **critical section** is a section of code in which processes access and modify shared data.
- **Mutual exclusion** is the act of ensuring that 1 or N processes are in a critical section at a time. Others must wait.
- **Critical Section Solution:**
    - **Mutual exclusion**, if a process is executing in the critical section, no other process can execute in the critical section.
    - **Progress,** if no process is in the critical section, and some process wish to enter the critical selection, the selection of the next process to enter the critical selection cannot be postponed indefinitely.
    - **Bounded Waiting**, a bound must exist between a request to enter the critical section, and the number of processes that enter before the requesting process before its request is granted.
- Three possibilities exist to implement mutual exclusion. They are **application, hardware,** and **OS.**
- **Application implemented Mutual Exclusion:**
    - Implemented by the programmer.
    - Inefficient and prone to error.
    - Relies on busy waiting
    - **Implementations:**
        - Use a lock (lock & unlock to enter and leave a critical section respectively)
        - Disable interrupts (supported by HW, no preemption when you enter a critical section)
            - **CONS:** Privileged operation, dangerous, only for a single processor. Slow.
- **Synchronization HW:**
    - **Test-And-Set (TAS)**: HW instruction tests and modifies the contents of a memory word atomically (in an arbitrary order). Therefore, only a single process can modify a variable at a time.
    - **Spin lock** occurs when a process spins at the entrance of CS waiting to be granted entry. It is busy waiting and is inefficient, and not fair.
    - **Compare-And-Swap**: Test if a value at the address is equal to the expected value. If so, update the memory with the new value. If not, do nothing.
    - **HW Advantages:** Single or multiple processes (shared memory), Simple & easy to verify, supports multiple critical sections
    - **HW Disadvantages:** Busy waiting is used, starvation and deadlock possible.

- **OS Synchronization:**
  - **Mutex locks:** A simple OS tool that allows a process to acquire() and release() a lock on a critical section. Calls must be **atomic.**
    - Advantage: Simple
    - Disadvantage: Busy waiting
  - **Semaphores:** A non-negative integer that has two valid operations.
    - **wait():** If sem > 0 then sem -= 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 very short time.
    - **Advantages:** Avoids busy waiting!
    - **Disadvantages:** Error prone & difficult to use.
  - **Monitors:** A language construct that includes synchronization. Only one process can entire the monitor at a time. (OS implementation could use a sem to implement or condition variables).
  - **Condition variables:** A variable where only two operations are allowed.
    - **cond.wait():** Invoking process is suspended until a cond.signal().
    - **cond.signal():** Wakes up a process that invoked cond.wait(). Otherwise does nothing.
    - **What occurs if a process invokes signal() if a process already suspended in wait()**
      - **Signal & Wait:** Signaling process waits for suspended process to leave the monitor or for another condition
      - **Signal & Continue:** Signaling process enters the monitor and suspended process waits for the process to leave the monitor or for another condition.
- **Classical IPC problems & solutions:**
  - **Bounded Buffer:** A producer and consumer process shared a bounded circular buffer. Producer puts items in, consumer removes items. How do you we prevent race conditions if these two processes run in parallel?
    - **Solution:** Three semaphores. Two are counting semaphores that keep track 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:** If a writer is waiting, prevent readers from entering. Waiting proceeds when no readers remain.
    - **Solution 2:** A writer can only enter when no readers are reading. Therefore, it must wait until no readers are active before entering. Requires 3 semaphores.
  - **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. Have odd pick up a specific side first and even the other. One semaphore per chopstick. If a chopstick is being used wait. Else signal and pick up the chopstick.

Ch8 – Deadlocks

- A **deadlock** occurs when a **cycle** occurs. A locked resource is wanted by a process that is held by another process that wants the resource the original process holds.

- **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 be released only voluntarily by the process holding it.
  - **Circular Wait:** There exits a set of processes such that P_0 is waiting for a resource held by P_1 which is for a resource held by P_{N-1} and P_N is waiting for a resource held by P_0.
- **Handling Deadlocks:**
  - **Deadlock prevention:** Ensure a deadlock will never occur (invalidate at least 1 of 4 characteristics).
  - **Deadlock avoidance:** Never allow the system to enter a deadlocked state.
  - **Ignore the problem:** Done by most OS's as the above are expensive ops.
- **Deadlock Prevention (Govern the ways a process can hold/release a process):**
  - **Mutual exclusion** not required for sharable resources. There is no deadlock. Must hold for non-sharable though…
  - **Hold & Wait:** must guarantee that whenever a process request a resource, it currently holds none.
    - Require a process to be allocated all its resources at once. Or only when it has no allocated resources.
    - Low resource utilization and possible starvation.
  - **Preemption:** If a process holding some resources request another resource and it must wait… Release all the resources it currently holds and put the process to sleep. Only wake it up when all previous and requested resources are available.
  - **Circular Wait:** Impose a total ordering of all resources (most common). Require each process request a resource in increasing order.
- **Deadlock avoidance:**
  - Requires the system have "in advance" information.
    - Each process declares max number of resources they may need. A deadlock avoidance algorithm determines if the system is in a safe state and/or a request allocation request can proceed.
  - **Safe State:** If there exists a sequence of **ALL** processes on the system such that a request can be satisfied by currently available resources and resources held by running processes that complete.
    - **Safe state** = no deadlocks. **Unsafe state** = possibility of deadlocks. **Avoidance** = ensure the system is always in a safe state.
  - **Avoidance algorithms:**
    - If only a single instance of a resource type. A **resource allocation** graph can be used to allocate to determine the state. Claim edges show that a process could request said resource.
    - If multiple instances of a resource type… Use **Banker's Algorithm**
- **Banker's Algorithm**:
  - **Available vector:** lists the number of available resources of each type
  - **Max matrix:** lists the max number of resources a process could request for each resource type.
  - **Allocation matrix:** lists the currently allocated resources for each process of each type.
  - **Need matrix:** the difference between the max and allocation matrix. This is how many resources of each type a process could request.

- **Safety Algorithm:**
  1. Let work & finish be vectors of length M & N respectively:
     a. Work = available
     b. Finish[i] = false for i = 0, 1, …, N – 1
  2. Find an i such that both
     a. Finish[i] = false
     b. $Need_i$ <= Work
  3. If no such i exists… go to step 4
     a. Work = work + allocation
     b. Add resource used by process to work. Imagine resources have been reclaimed.
     c. Finish[i] = true
  4. If finish[i] = true for all i… then the system is in a safe state. Else it is unsafe.
- **Resource Request Algorithm for P_i**
  1. $Request_i$ = request vector for process P_i
  2. If $request_i$ <= need go to step 3, else error as max claim exceeded
  3. If $request_i$ <= available go to step 4, else it must wait
  4. Pretent to allocate requested resources
     a. Available = Available – request
     b. Allocation = Allocation + request
     c. Need = need – request
  5. Run the safety algorithm, if a safe sequence exits allocate the resources… Else, the process must wait

Study resource allocation graphs and cycles…

- **Resource Allocation Graphs:**
  - If a cycle exists with only 1 resource instance deadlock
  - If a cycle exists with >1 resource instance deadlock possible.