CSC 139 – Midterm 2 Study Guide

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.