CSI 450 — Operating Systems, Fall 2009

Review Sheet #3

- Chapter 6 - Process Synchronization
  - producer/consumer, bounded buffer, concurrent execution (6.1: 225–227)
  - The critical section problem (6.2: 227)
  - The critical section solution (mutual exclusion, progress, bounded waiting) (6.2: 227-228)
  - race conditions, preemptive kernel vs nonpreemptive kernel (6.2: 228–229)
  - Peterson's Solution (6.3: 229–231)
  - atomic instructions,TestAndSet, Swap (6.4: 231–234)
  - semaphores, counting, binary, mutex, synchronization (6.5: 234–235)
  - implementation- busy waiting, spinlock, or blocking (6.5.2: 235–238)
  - deadlock and starvation (6.5.3: 238)
  - classic problems - bounded buffer, dining philosophers, sleeping barber (6.6: 239–244)
- Chapter 7 - Deadlock
  - Why is deadlock a problem? (p283–285)
  - Four simultaneous conditions for deadlock to occur (7.2: 285-287)
    1. Mutual exclusion
    2. No preemption
    3. Hold and wait
    4. Circular Wait
  - System Resource Allocation Graph (7.2.2: 287–289)
    * A set of vertexes V and a set of edges E
    * Vertexes contain active processes $P = \{P_1...P_n\}$ (denoted as circles) and system resources $R = \{R_1...R_n\}$ (denoted as squares).
    * Each instance of a resource is denoted with a separate dot inside that resources' square.
    * Edges contain request edges and assignment edges. Request edges point to the resources' square. Assignment edges point from a particular resource instance dot.
    * $P_i \rightarrow R_j$ is a request edge. It means that a processes P_i has requested resource R_j.
    * $R_j \rightarrow P_i$ is an assignment edge. It means that a resource R_j has been allocated to a process P_i

- * A graph with no cycle means that there is no deadlock. A graph with a cycle means deadlock may exist. In single instance resource systems, a cycle means that dead lock does exist.

– Handling Deadlock (7.3: 290–291) - Deadlock Prevention, Deadlock Avoidance, Deadlock Detection and Recovery, Ignoring Deadlock

– Deadlock Prevention (7.4: 291–294) - prevent deadlock by removing one of the four essential components.

  1. mutual exclusion - useful but cannot be done for all resources
  2. hold and wait - (i) request only when it has none or (ii) access all resources before execution may begin
  3. no preemption - (i) implicitly release all resources when a resource requested is not available or (ii) take resources from processes that have them allocated and that are also waiting
  4. circular wait - (i) Impose a strict order on resources and force each process to request resources in that order. Define a function $F(R_i)$ which returns this order.

– Deadlock Avoidance (7.5: 294–296) - Use the resource allocation graph to avoid deadlock at all times. (safe state, safe sequence)

– Deadlock avoidance with single instance resources (7.5.2: 296–297)

  * Add a claim edge $P_i - - > R_j$ to symbolize that P_i may request R_j in the future.
  * A process which begins execution with no resource will add claim edges for every resource it may acquire in the future.
  * With this addition, we avoid deadlock by allowing R_j to be allocated to P_i only if this results in a graph with no cycles.

– Deadlock avoidance with multiple instance resources (7.5.3: 298–300) - Banker's Algorithm

  * n processes, m resources. Set up four data structures: $Available[m]$, $Max[n][m]$, $Allocation[n][m]$, and $Need[n][m]$.
  * Safety Algorithm (p298–299)
    1. $Work[m]$, $Finish[n]$. $\forall j \in m, Work[j] = Available[j]$,
       $\forall i \in n, Finish[i] = false$
    2. Find an i such that
       (a) $Finish[i] = false$ and
       (b) $\forall j \in m, Need[i][j] \leq Work[j]$
       If no such i exists, goto Step 4, else goto Step 3 with i
    3. Set $Finish[i] = true$ and $\forall j \in m, Work[j] = Work[j] + Allocation[i][j]$
       Go to Step 2.
    4. If $\forall i \in n, Finish[i] == true$, then the system is in a safe state and the safe sequence is the order in which each i was found. Otherwise, the system is unsafe.

    - Resource Request Algorithm (p299)
    - When a process P_i wants resources, it will send a $Request[m]$. This request is granted if:
        1. If $\forall j \in m, Request[j] \leq Need[i][j]$ Go To Step 2. Otherwise raise an error since P_i has exceeded its claimed maximum needed.
        2. If $\forall j \in m, Request[j] \leq Available[j]$ Go to Step 3. Otherwise, P_i must wait since there are not enough resources ready yet.
        3. Do a temporary allocation of resources to P_i by doing the following:
            (a) $\forall j \in m, AvailableTemp[j] = Available[j] - Request[j]$
            (b) $\forall j \in m, AllocationTemp[i][j] = Allocation[i][j] + Request[j]$
            (c) $\forall j \in m, NeedTemp[i][j] = Need[i][j] - Request[i][j]$

            With these temporary data structures, run the Safety Algorithm. If the system is determined to be in a safe state, then these arrays become permanent and P_i is allocated the resources it desires. Otherwise, if the new state is unsafe then P_i must wait to be allocated its resources.
- Deadlock Detection and Recovery (7.6: 301–304)
    - For single instances - uses a resource-allocation graph and find cycles
    - For multiple instances, (when a process requests resources, run a modified safety algorithm. This modified safety algorithm will use $Request[n][m]$ for the currently requested resources of each process in place of $Need[n][m]$. If at Step 4, $Finish[i] == false$ then deadlock has been detected. In particular, the process P_i where $Finish[i] == false$ is a deadlocked one.
    - Recovery from Deadlock (7.7: 304–306) - process termination or resource preemption
    - process termination: abort all deadlocked processes or abort one process at a time until deadlock is eliminated.
    - resource preemption: (i) select a victim based on number of resources held by a deadlocked process, amount of time the process has executed, and the number of times the process was already chosen as a victim, (ii) rollback the process to a safe state, (iii) ensure starvation does not occur