Announcements

- Reading
  - Today
    - 8.1-8.3, 8.6 (6th Ed)
    - 7.1-7.3, 7.6 (8th Ed)

- Project #2 is due next Tuesday at 5:00 PM (3/4/14)
- Midterm #1 is next Thursday (3/6/16) in class
- P1 grading – still working on a fix for the submit bug
  - Will email those who lost style points soon

Sample Synchronization Problem

- Class Exercise:
  - **CMSC 412 Midterm #1 (Spring 1998)** Q#3
- Went over master solution

- Variables:

Semaphore mutex = 1

Semaphore writer = 0

Semaphore reader = 0

int nReader = 0

int nWriter = 0

int wReader = 0

int wWriter = 0

- Writers execute this code:

```
while (1) {
        P(mutex);
        if (nReader + wReader + nWriter == 0)  {
                nWriter++;
                V(mutex);
        } else {
                wWriter++;
                V(mutex);
                P(writer);
        }
         // Write operation;
        P(mutex);
        NWriter = 0;
        If (wReaders > 0) {
                Temp = min(wReaders,5)
                for i = 1 to temp {
                        V(readers)
                        nReaders++;
                        wReaders--;
          }
        } else if (wWriters > 0) {
                wWriters--;
                nWriters++; V(writer);
          } V(mutex);
  }
```

- Readers execute this code:

```
while (1) {
        P(mutex)
        if (nWriters + wWriter == 0 & nReader < 5) {
                 nReaders++;
                V(mutex);
        } else {
                wReaders++;
                V(mutex);
                P(reader);
        }
        // Read operation;
        P(mutex);
        nReaders--;
        if (wWriters > 0 & nReaders == 0) {
                wWriters--;
                nWriters++;
                V(writer);
        } else if (wReaders > 0 & wWriters == 0) {
                nReaders++;
                wReaders--;
                V(reader);
        }
        V(mutex);
}
```

Deadlocks

- System contains finite set of resources
  - memory space
  - printer
  - tape
  - file
  - access to non-reentrant code
- Process requests resource before using it, must release resource after use
- Process is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set

Formal Deadlocks

- 4 *necessary* deadlock conditions:
  - Mutual exclusion - at least one resource must be held in a non-sharable mode, that is, only a single process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource is released
  - Hold and wait - There must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently held by other processors

Formal Deadlocks

  - No preemption: Resources cannot be preempted; a resource can be released only voluntarily by the process holding it, after that process has completed its task
  - Circular wait: There must exist a set {P0,...,Pn} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource held by P2 etc.
- Note that these are not sufficient conditions

Detecting Deadlock

Work is a vector of length m (resources)

Finish is a vector of length n (processes)

- Allocation is an n x m matrix indicating the number of each resource type held by each process
- Request is an m x n matrix indicating the number of additional resources requested by each process

1. Work = Available;

   if Allocation[i] != 0 Finish = false else Finish = true;

   This is the difference from the Banker's algorithm.

2. Find an *i* such that Finish[i] = false and $Request_i$ <= Work if no such i, go to 4
3. Work += Allocation ; Finish[i] = true; goto step 2
4. If Finish[i] = false for some i, system is in deadlock

Note: this requires m x n^2 steps

Recovery from deadlock

- Must free up resources by some means
- Process termination
  - kill all deadlocked processes
  - select one process and kill it
    - must re-run deadlock detection algorithm again to see if it is freed.
- Resource Preemption
  - select a process, resource and de-allocate it
  - rollback the process
    - needs to be reset the process to a safe state
    - this requires additional state
  - starvation
    - what prevents a process from never finishing?

Deadlock Prevention

- Ensure that one (or more) of the necessary conditions for deadlock do not hold
- Hold and wait
  - guarantee that when a process requests a resource, it does not hold any other resources
  - Each process could be allocated all needed resources before beginning execution
  - Alternately, process might only be allowed to wait for a new resource when it is not currently holding any resource

Deadlock Prevention

- Mutual exclusion
  - Sharable resources do not require mutually exclusive access and cannot be involved in a deadlock.
- Circular wait
  - Impose a total ordering on all resource types and make sure that each process claims all resources in increasing order of resource type enumeration
- No Premption
  - virutalize resources and permit them to be prempted. For example, CPU can be prempted.

Deadlock Avoidance

- Require additional information about how resources are to be requested - decide to approve or disapprove requests on the fly
- Assume that each process lets us know its maximum resource request
- Safe state:
  - system can allocate resources to each process (up to its maximum) in *some order* and still avoid a deadlock
  - A system is in a safe state if there exists a *safe sequence*

Safe Sequence

- Sequence of processes $<P_1, .. P_n>$ is a safe sequence if for each P_i, the resources that P_i can request can be satisfied by the currently available resources plus the resources held by all P_j, $j<i$
- If the necessary resources are not immediately available, P_i can always wait until all P_j, $j<i$ have completed

Banker's Algorithm

- Each process must declare the maximum number of instances of each resource type it may need
- Maximum can't exceed resources available to system
- Variables:

  n is the number of processes

  m is the number of resource types

  – Available - vector of length m indicating the number of available resources of each type
  – Max - n by m matrix defining the maximum demand of each process
  – Allocation - n by m matrix defining number of resources of each type currently allocated to each process
  – Need: n by m matrix indicating remaining resource needs of each process

- Work is a vector of length m (resources)
- Finish is a vector of length n (processes)

1. Work = Available; Finish = false
2. Find an *i* such that Finish[i] = false and $Need_i$ <= Work if no such i, go to 4
3. Work += $Allocation_i$; Finish[i] = true; goto step 2
4. If Finish[i] = true for all i, system is in a safe state

all elements in the vector are <=

Note this requires m x n^2 steps