Announcements

- Midterm is next Thursday
  – Covers through today's lecture
- Reading
  – Chapter 7 – can skip 7.7 & 7.9
  – Today Chapter 8
- Project #2 will be available on the web
- Suggested problems:
  – 7.1, 7.2, 7.6, 7.8, 7.9, 7.15, 7.18

Writers Have Priority

reader

```
repeat
        P(z);
                P(rsem);
                P(x);
                        readcount++;
                        if (readcount == 1) then
                                            P(wsem);
                V(x);
                V(rsem);
        V(z);
        readunit;
        P(x);
                readcount- -;
                if readcount == 0 then
                                    V (wsem)
        V(x)
forever
```

writer

```
repeat
        P(y);
                writecount++:
                if writecount == 1 then
                                        P(rsem);
        V(y);
        P(wsem);
        writeunit
        V(wsem);
        P(y);
                writecount--;
                if (writecount == 0) then
                                        V(rsem);
        V(y);
forever;
```

Notes on readers/writers with writers getting priority

Semaphores x,y,z,wsem,rsem are initialized to 1

```
P(z);
      P(rsem);
      P(x);
            readcount++;
            if (readcount==1) then
                                P(wsem);
      V(x);
      V(rsem);
V(z);
```

readers queue up on semaphore z; this way only a single reader queues on rsem. When a writer signals rsem, only a single reader is allowed through

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

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

all elements in the vector are <=

- 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

Note this requires m x n^2 steps

Banker's Algorithm - Example

Three resources: A, B, C (10, 5, 7 instances each)

Consider the snapshot of the system at this time

	Alloc	Max	Avail	Need (Max - alloc)
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2	7 4 3
P1	2 0 0	3 2 2		1 2 2
P2	3 0 2	9 0 2		6 0 0
P3	2 1 1	2 2 2		0 1 1
P4	0 0 2	4 3 3		4 3 1

System is in a safe state, since the sequence <P1, P3, P4, P2, P0> satisfy the safety criteria.

Resource Request Algorithm

(1) If $Request_i$ <= $Need_i$ then goto 3

- otherwise - the process has exceeded its maximum claim

(2) If $Request_i$ <= Available then goto 3

- otherwise process must wait since resources are not available

(3) Check request by having the system pretend that it has allocated the resources by modifying the state as follows:

- Available =Available - $Request_i$
- Allocation = Allocation + $Request_i$
- $Need_i = Need_i - Request_i$

- Find out if resulting resource allocation state is safe, otherwise the request must wait.