Announcements

- Reading chapter 6 (6.7) and chapter 7 (7.1-7.4)
- Midterm #1 is March 5 in class
- Late Policy for programs
  - no late work will be accepted
  - illness and family emergency will be considered on a case by case basis
Implementing Semaphores

- **declaration**
  
  ```
  type semaphore = record
    value: integer;
    L: list of process;
  end;
  ```

- **P(S): S.value = S.value -1**
  
  ```
  if S.value <= 0 then {
    add this process to S.L
    block;
  }
  ```

- **V(S): S.value = S.value +1**
  
  ```
  if S.value > 0 then {
    remove process P from S.L
    wakeup(P);
  }
  ```
Readers/Writers Problem

- Data area shared by processors
- Some processors read data, other processors can read or write data
  - Any number of readers may simultaneously read the data
  - Only one writer at a time may write
  - If a writer is writing to the file, no reader may read it
- Two of the possible approaches
  - readers have priority or writers have priority
Readers have Priority

reader()
{
    repeat
        P(x);
        readcount = readcount + 1;
        if readcount = 1 then P (wsem);
        V(x);
        READUNIT;
        P(x);
        readcount = readcount - 1;
        if readcount = 0 V(wsem);
        V(x);
        forever
    }

writer()
{
    repeat
        P(wsem);
        WRITEUNIT;
        V(wsem);
        forever
    }
Comments on Reader Priority

- semaphores x,wsem are initialized to 1
- note that readers have priority - a writer can gain access to the data only if there are no readers (i.e. when readcount is zero, signal(wsem) executes)
- possibility of starvation - writers may never gain access to data
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 \( x, y, z, w_{\text{sem}}, r_{\text{sem}} \) are initialized to 1

readers queue up on semaphore \( z \); this way only a single reader queues on \( r_{\text{sem}} \). When a writer signals \( r_{\text{sem}} \), only a single reader is allowed through

```c
P(z);
P(r_{\text{sem}});
P(x);
readcount++;
if (readcount==1) then
  P(w_{\text{sem}});
V(x);
V(r_{\text{sem}});
V(z);
```
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 \{P_0,...,P_n\} of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource held by P_2 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**
  - Virtualize resources and permit them to be preempted. For example, CPU can be preemted.
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