Announcements

- **Office hours**
  - W office hour will be 10-11 not 11-12 starting this week

- **Midterm is next Tuesday**
  - Covers through lecture on Thursday

- **Project #2 is available on the web**
Using Test and Test for Mutual Exclusion

repeat
    while test-and-set(lock);  // critical section
    lock = false;  // non-critical section
until false;

• bounded waiting time version

repeat
    waiting[i] = true;
    key = true;
    while waiting[i] and key
        key = test-and-set(lock);
    waiting[i] = false;  // critical section
    j = (i + 1) % n
    while (j != i) and (!waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;  // non-critical section
    else
        waiting[j] = false;
    until false;

Note: no priority based on wait time
wait until released or no one busy
look for a waiting process
no process waiting
release process j
Semaphores

- **getting critical section problem correct is difficult**
  - harder to generalize to other synchronization problems
  - Alternative is semaphores

- **semaphores**
  - integer variable
  - only access is through atomic operations

- **P (or wait)**
  
  ```
  while s <= 0;
  s = s - 1;
  ```

- **V (or signal)**
  
  ```
  s = s + 1
  ```

- **Two types of Semaphores**
  - Counting (values range from 0 to n)
  - Binary (values range from 0 to 1)
Using Semaphores

- critical section
  
  repeat
  
  P(mutex);
  
  // critical section
  
  V(mutex);
  
  // non-critical section
  
  until false;

- Require that Process 2 begin statement S2 after Process 1 has completed statement S1:

  Process 2
  
  S1
  
  V(synch)
  
  Process 1
  
  P(synch)
  
  S2
Implementing semaphores

- Busy waiting implementations
- Instead of busy waiting, process can block itself
  - place process into queue associated with semaphore
  - state of process switched to waiting state
  - transfer control to CPU scheduler
  - process gets restarted when some other process executes a signal operations
Implementing Semaphores

- **declaration**

  ```
  type semaphore = record
    value: integer = 1;
    L: FIFO 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);
  }
  ```

*Revised from class :-(

*Can be neg, if so, indicates how many waiting

*Bounded waiting!!
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, wsem, rsem are initialized to 1

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

$P(z);$
$P(rsem);$
$P(x);$
$readcount++;$
$if (readcount==1) then$
$P(wsem);$
$V(x);$
$V(rsem);$
$V(z);$
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, \ldots, 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 preempted.