Announcements

- Program #1
  - Due on Th 9:00 AM
- Midterm #1
  - Thursday of next week (March 6th)
- Reading
  - Chapter 7 (this whole week)

Synchronization Hardware

- If it's hard to do synchronization in software, why not do it in hardware?
- Disable Interrupts
  - works, but is not a great idea since important events may be lost.
  - doesn't generalize to multi-processors
- test-and-set instruction
  - one atomic operation
    - executes without being interrupted
  - operates on one bit of memory
  - returns the previous value and sets the bit to one
- swap instruction
  - one atomic operation
  - swap(a,b) puts the old value of b into a and of a into b

Using Test and Set for Mutual Exclusion

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

Note: no priority based on wait time

- 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;
        else
             waiting[j] = false;
        // non-critical section
until false;
```

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:

```
semaphore synch = 0;
Process 1
    S1
    V(synch)
Process 2
    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;
```

Revised from class :-(

- P(S):

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

Can be neg, if so, indicates how many waiting

- V(S):

```
S.value = S.value+1
if S.value <= 0 then {
        remove process P from S.L
        wakeup(P);
}
```

Bounded waiting!!

Readers/Writers Problem

- Data area shared by processors
- Some processes read data, others write data
  - Any number of readers my 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

```
Semaphore wsem = 1, x = 1;
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

```
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