Announcements

- Program #2 handouts are on the web page
- Reading chapter 6 (6.3)
Producer-consumer pair

- producer creates data and sends it to the consumer
- consumer read the data and uses it
- examples: compiler and assembler can be used as a producer consumer pair

Buffering
- processes may not produce and consume items one by one
- need a place to store produced items for the consumer
  - called a buffer
- could be fixed size (bounded buffer) or unlimited (unbounded buffer)
Message Passing

- What happens when a message is sent?
  - sender blocks waiting for receiver to receive
  - sender blocks until the message is on the wire
  - sender blocks until the OS has a copy of the message
  - sender blocks until the receiver responds to the message
    - sort of like a procedure call
    - could be expanded into a remote procedure call (RPC) system

- Error cases
  - a process terminates:
    - receiver could wait forever
    - sender could wait or continue (depending on semantics)
  - a message is lost in transit
    - who detects this? could be OS or the applications

- Special case: if 2 messages are buffered, drop the older one
  - useful for real-time info systems
Signals (UNIX)

- provide a way to convey one bit of information between two processes (or OS and a process)
- types of signals:
  - change in the system: window size
  - time has elapsed: alarms
  - error events: segmentation fault
  - I/O events: data ready
- are like interrupts
  - a processes is stopped and a special handler function is called
- a fixed set of signals is normally available
Producer-consumer: shared memory

- Consider the following code for a producer
  repeat
    ....
    produce an item into nextp
    ...
    while counter == n;
    buffer[in] = nextp;
    in = (in+1) % n;
    counter++;
  until false;

- Now consider the consumer
  repeat
    while counter == 0;
    nextc = buffer[out];
    out = (out + 1) % n;
    counter--;
    consume the item in nextc
  until false;

- Does it work? Answer: NO!
Problems with the Producer-Consumer Shared Memory Solution

- Consider the three address code for the counter
  
<table>
<thead>
<tr>
<th>Counter Increment</th>
<th>Counter Decrement</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reg1 = counter</code></td>
<td><code>reg2 = counter</code></td>
</tr>
<tr>
<td><code>reg1 = reg1 + 1</code></td>
<td><code>reg2 = reg2 - 1</code></td>
</tr>
<tr>
<td><code>counter = reg1</code></td>
<td><code>counter = reg2</code></td>
</tr>
</tbody>
</table>

- Now consider an ordering of these instructions
  
  | T_0 producer | reg1 = counter { reg1 = 5 } |
  | T_1 producer | reg1 = reg1 + 1 { reg1 = 6 } |
  | T_2 consumer | reg2 = counter { reg2 = 5 } |
  | T_3 consumer | reg2 = reg2 - 1 { reg2 = 4 } |
  | T_4 producer | counter = reg1 { counter = 6 } |
  | T_5 consumer | counter = reg2 { counter = 4 } |

This should be 5!
Defintion of terms

- **Race Condition**
  - Where the order of execution of instructions influences the result produced
  - Important cases for race detection are shared objects
    - counters: in the last example
    - queues: in your project

- **Mutual exclusion**
  - only one process at a time can be updating shared objects

- **Critical section**
  - region of code that updates or uses shared data
    - to provide a consistent view of objects need to make sure an update is not in progress when reading the data
  - need to provide mutual exclusion for a critical section
Critical Section Problem

- Processes must
  - request permission to enter the region
  - notify when leaving the region

- Protocol needs to
  - provide mutual exclusion
    - only one process at a time in the critical section
  - ensure progress
    - no process outside a critical section may block another process
  - guarantee bounded waiting time
    - limited number of times other processes can enter the critical section while another process is waiting
  - not depend on number or speed of CPUs
    - or other hardware resources
Critical Section (cont)

- May assume that some instructions are atomic
  - typically load, store, and test word instructions

- Algorithm #1 for two processes
  - use a shared variable that is either 0 or 1
  - when $P_k = k$ a process may enter the region

```
repeat
  (while turn != 0);
  // critical section
  turn = 1;
  // non-critical section
until false;
```

```
repeat
  (while turn != 1);
  // critical section
  turn = 0;
  // non-critical section
until false;
```

- this fails the progress requirement since process 0 not being in the critical section stops process 1.