Announcements

● Program #2 handouts were provided
● 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 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 to provide 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 two 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 process 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

```c
repeat
    ....
    produce an item into nextp
    ...
    while counter == n;
    buffer[in] = nextp;
    in = (in+1) % n;
    counter++;
until false;
```

● Now consider the consumer

```c
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>Instruction</th>
<th>Register Assignment</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counter Increment</td>
<td>reg₁ = counter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>reg₁ = reg₁ + 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>counter = reg₁</td>
<td></td>
</tr>
<tr>
<td>Counter Decrement</td>
<td>reg₂ = counter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>reg₂ = reg₂ - 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>counter = reg₂</td>
<td></td>
</tr>
</tbody>
</table>

● Now consider an ordering of these instructions

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Instruction</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₀</td>
<td>producer</td>
<td>reg₁ = counter</td>
<td>{ reg₁ = 5 }</td>
</tr>
<tr>
<td>T₁</td>
<td>producer</td>
<td>reg₁ = reg₁ + 1</td>
<td>{ reg₁ = 6 }</td>
</tr>
<tr>
<td>T₂</td>
<td>consumer</td>
<td>reg₂ = counter</td>
<td>{ reg₂ = 5 }</td>
</tr>
<tr>
<td>T₃</td>
<td>consumer</td>
<td>reg₂ = reg₂ - 1</td>
<td>{ reg₂ = 4 }</td>
</tr>
<tr>
<td>T₄</td>
<td>producer</td>
<td>counter = reg₁</td>
<td>{ counter = 6 }</td>
</tr>
<tr>
<td>T₅</td>
<td>consumer</td>
<td>counter = reg₂</td>
<td>{ counter = 4 }</td>
</tr>
</tbody>
</table>

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

\[
\text{repeat} \\
\text{\hspace{1em} (while turn != 0);} \\
\text{\hspace{2em} // critical section} \\
\text{\hspace{2em} turn = 1;} \\
\text{\hspace{2em} // non-critical section} \\
\text{\hspace{1em} until false;} \\
\]

\[
\text{repeat} \\
\text{\hspace{1em} (while turn != 1);} \\
\text{\hspace{2em} // critical section} \\
\text{\hspace{2em} turn = 0;} \\
\text{\hspace{2em} // non-critical section} \\
\text{\hspace{1em} until false;} \\
\]

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