Announcements

- **Program #1**
  - Due in one week (2)

- **Reading**
  - Chapter 7
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
Signals

Signal Handler Table

SetSigAction(sig, handler)

SigAlarmHandler
{
}

SigIOHandler
{
}
Shared Memory

- Like Threads, but only part of memory shared
- Allows communication without needing kernel action
  - Kernel calls setup shared region
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?
  NO!
Problems with the Producer-Consumer Shared Memory Solution

- Consider the three address code for the counter
  
  **Counter Increment**
  
  \[ \text{reg}_1 = \text{counter} \]
  
  \[ \text{reg}_1 = \text{reg}_1 + 1 \]
  
  \[ \text{counter} = \text{reg}_1 \]
  
  **Counter Decrement**
  
  \[ \text{reg}_2 = \text{counter} \]
  
  \[ \text{reg}_2 = \text{reg}_2 - 1 \]
  
  \[ \text{counter} = \text{reg}_2 \]

- Now consider an ordering of these instructions

  \[ T_0 \quad \text{producer} \quad \text{reg}_1 = \text{counter} \quad \{ \text{reg}_1 = 5 \} \]
  
  \[ T_1 \quad \text{producer} \quad \text{reg}_1 = \text{reg}_1 + 1 \quad \{ \text{reg}_1 = 6 \} \]
  
  \[ T_2 \quad \text{consumer} \quad \text{reg}_2 = \text{counter} \quad \{ \text{reg}_2 = 5 \} \]
  
  \[ T_3 \quad \text{consumer} \quad \text{reg}_2 = \text{reg}_2 - 1 \quad \{ \text{reg}_2 = 4 \} \]
  
  \[ T_4 \quad \text{producer} \quad \text{counter} = \text{reg}_1 \quad \{ \text{counter} = 6 \} \]
  
  \[ T_5 \quad \text{consumer} \quad \text{counter} = \text{reg}_2 \quad \{ \text{counter} = 4 \} \]

This should be 5!
Definition 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

- **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

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

```c
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.
Critical Section (Algorithm 2)

- Keep an array of flags indicating which processes want to enter the section

```c
bool flag[2];

repeat
  flag[i] = true;
  while (flag[j]);

// critical section

flag[i] = false;

// non-critical section
until false;
```

- This does NOT work either!
  - possible to have both flags set to 1

Both processes could be here at the same time
Critical Section (Algorithm 3)

- Combine 1 & 2

```c
bool flag[2];
int turn;

repeat
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

    // critical section

    flag[i] = false;

    // non-critical section
    until false;
```

- This one does work! Why?
Critical Section (many processes)

- What if we have several processes?
- One option is the Bakery algorithm

```plaintext
bool choosing[n];
integer number[n];

choosing[i] = true;
number[i] = max(number[0],..number[n-1])+1;
choosing[i] = false;
for j = 0 to n-1
    while choosing[j];
    while number[j] != 0 and ((number[j], j) < number[i],i);
end
// critical section
number[i] = 0
```
Bakery Algorithm - explained

- When a process wants to enter critical section, it takes a number
  - however, assigning a unique number to each process is not possible
    - it requires a critical section!
  - however, to break ties we can use the lowest numbered process id

- Each process waits until its number is the highest one
  - it can then enter the critical section

- provides fairness since each process is served in the order they requested the critical section