Announcements

- **Program #1**
  - Is one the Web
  - Due in Friday 3/5/10 at 6:00 PM

- **Reading**
  - Chapter 6 (8th Ed) or Chapter 7 (6th Ed)
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 process 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
{
}

CMSC 412 – S10 (lect 7)
Shared Memory

- Like Threads, but only part of memory shared
- Allows communication without needing kernel action
  - Kernel calls setup shared region

Process 1

Process 2

Shared Region
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?
  - 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

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

```cpp
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
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 lowest one
  - it can then enter the critical section
- provides fairness since each process is served in the order they requested the critical section
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 (depending on HW)
  - 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;

• 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;

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:**

  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;
  ```

- **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);
  }
  ```

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

*Bounded waiting!!*