Announcements

- Reading chapter 6 (6.4 and 6.5)
- Midterm #1 is March 5 in class
- Late Policy for programs
  - no late work will be accepted
  - illness and family emergency will be considered on a case by case basis
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.
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

```cpp
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
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 Test 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
  ```
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:**
  
  ```
  Process 2
  S1
  V(synch)
  Process 1
  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