Synchronization

Last lecture:
- Mutexes
  - Disable interrupts
  - Spinlock
    - Both are low level, and inefficient when it comes to application programming
    - Can waste time

Example:

<table>
<thead>
<tr>
<th>Lock</th>
<th>Owner</th>
<th>Queue</th>
</tr>
</thead>
</table>

The lock is a spinlock.
Owner is a pointer to the thread which holds the lock.
Queue is a queue of threads waiting to obtain the lock.

**m.lock()**: spin_lock(m.lock)
if owner = = null
  owner = me
else
  add me to the queue
  temp = owner
  spin_unlock(m.lock)
if temp != me
  sleep

**m.unlock()**: spin_lock(m.lock)
if q is empty
  owner = null
else
  owner = pop(queue)
  wake owner
  spin_unlock(m.lock)
Classical Problems in Synchronization

The Bounded Buffer Problem:
→ writer(s) call put() → buffer → get() called by reader(s)
→ There is a circular buffer that doesn't require locking:
  → writer modifies a head pointer
  → reader modifies a tail pointer that trails the head pointer
  → there is only one reader and one writer
→ Example Code:
  int head, tail;
  item buf[N];
  while (head + 1) mod N = = tail 
          ; (do nothing)
  buffer[head] = item;
  head = N + 1 mod N
→ This doesn't work for multiple readers/writers

Solution:
Semaphores: High level locking mechanisms (created in programs or provided by OS).
  Binary Semaphore: Essentially equivalent to a mutex
  Counting semaphore: rather than having only a 0 or 1 lock value, can have
    → a counter value (while counter > 0, threads don't block)
    → a wait() method (also known as p(), lock(), down()) which decreases the counter
      → causes thread to block when counter already = = 0
    → a signal method (aka v(), unlock(), up()) which increases the counter

Semaphore Example:
→ Semaphore S = new Semaphore(3);

This is a perfect fit for the Bounded Buffer Problem.

The numbers on the left of the image represent the Semaphore's count value. When it reaches zero, Threads requesting a lock must block and wait for a different lock holder to return the lock.
Mutex m;
Semaphore spaces(n), values(o);

get(): wait(values);
    lock(m);
    val = buf[tail];
    tail = tail + 1 mod n;
    unlock(m);
    signal(spaces);

put(): wait(spaces);
    lock(m);
    buf[head] = val;
    head = head + 1 mod n;
    unlock(m);
    signal(values);

Deadlocks:

    Thread 1                Thread 2
    lock(A);                       lock(B);
    lock(B);                       lock(A); ← At this point, Thread 1 requires Thread 2
    ......                      ...... to signal that it is done with B before it can
    ......                      ...... continue. However, Thread 2 needs Thread 1
    ......                      ...... to do the same regarding A before it can do so.
    ......                      ...... As a result both threads will remain blocked forever.
    ......                      ......
    unlock(B);    unlock(A); ← This code will never run.

The Reader/Writer Problem

The problem: Protecting resources that can have multiple readers simultaneously,
but only one writer at a time.

Methods necessary:
→ read_unlock();
→ read_lock();
→ write_unlock();
→ write_lock();

Implementation:
mutex m; int count; Semaphore wlock(1);

read_lock(): lock(m);
    if count = 0
        wait(wlock);
    count++;
    unlock(m);
read_unlock(): lock(m);
    if count = 0
        count--;
    signal(wlock);
    unlock(m);

write_lock(): wait(wlock);
write_unlock(): signal(wlock);
The 3 Rules of Thumb

Rule One:
→ Do not read shared variables outside of a mutex.
→ Use a local copy of the variable instead.

Rule Two:
→ Rank locks to avoid deadlock (sometimes...)
   → If you have 3 or more different locks, e.g. X, H, M, L
      Rank them, so that they need to be acquired in order, ie:
      H → L → X → M