Today’s Plan

Upcoming:
- Assignment 2 posted

Today’s topics:
- From last time:
  - Semaphores
- Producer/Consumer with Semaphores
- Semaphore Implementation

Last time:
- Software solutions to the CS problem
- Hardware solutions
SEM full = 0;  (# of items available for consumption)
SEM empty = N;  (# of empty buffer entries)
SEM mutex = 1;
Producer:  while (1) {
    -produce new item
    P(empty); (make sure we have at least 1 empty buffer)
P(mutex); (to guard against concurrent buffer manipulation)
    -add the new item to the buffer
    V(mutex); (signal OK for buffer access by consumer)
    V(full); (records addition of item to buffer)
}  
Consumer:  while(1) {
    P(full); (wait until at least 1 buffer is filled)
P(mutex); (wait for concurrency OK from producer)
    -remove item from the buffer
    V(mutex); (signal concurrency OK to producer)
    V(empty); (record the addition of 1 empty buffer)
    -consume item
}
What would happen if, on the producer, we reversed the lines "produce new item" and "P(empty)"?

What would happen (again in the producer) if we reversed the lines "P(empty)" and "P(mutex)"?
Semaphore Implementation

- Typically, semaphore operations are implemented within the operating system itself.

- Instead of a busy wait (while S <= 0;), the operating system can block the process
  - Saves the CPU for the use of other processes

- A process executing a V may cause another previously blocked process to be put on the ready queue.
Semaphore Implementation

typedef struct sem {
    int value;
    list_of_processes plist; (processes blocked on this semaphore)
} semaphore;

P(S):
    S.value--;
    if (S.value < 0) {
        add this process to S.plist;
        block;
    }

V(S):
    S.value++;
    if (S.value <= 0) {
        get a process P from S.plist;
        wakeup(P);
    }

How to ensure P and V are atomic?

- uni-processor: disable interrupts (P & V are fairly quick)
- multi-processor: software or hardware solutions
- busy wait is negligible for P & V