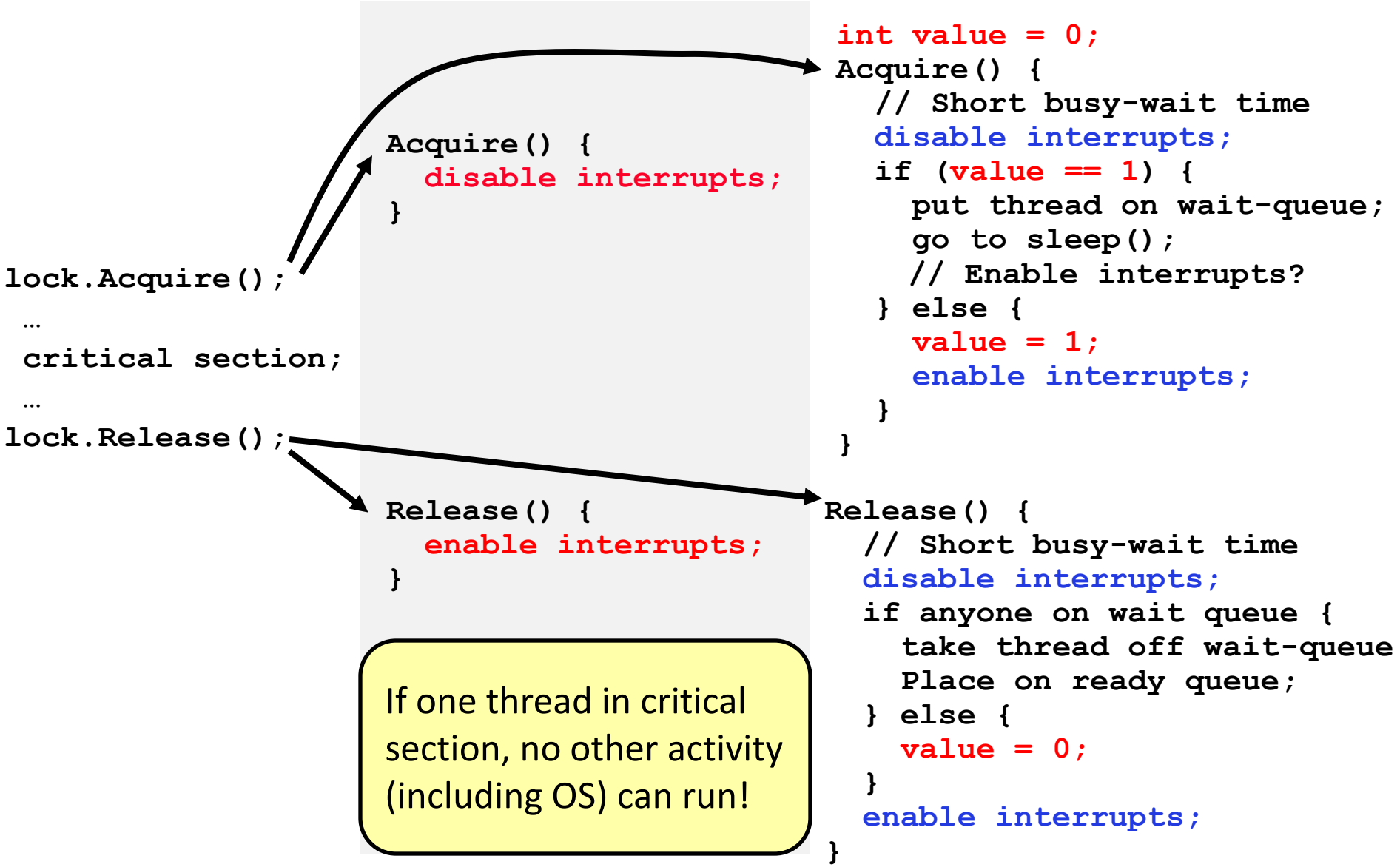
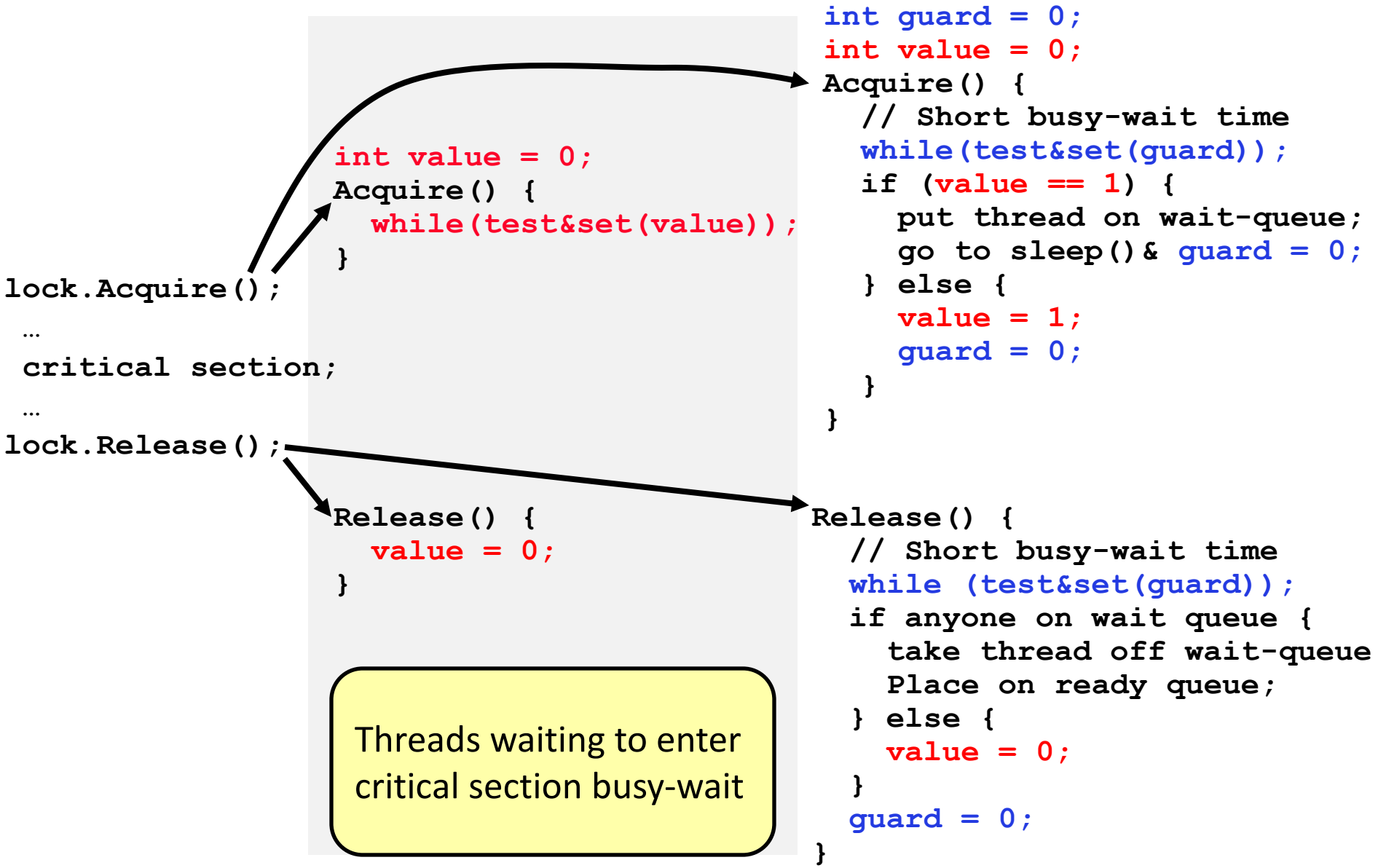
CSE150
Operating Systems
Lecture 8

Semaphores and Monitors

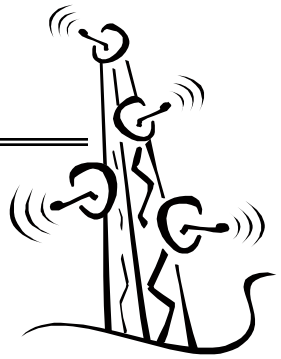
Review: Locks by Disabling Interrupts



Review: Locks using test&set



Communication & Coordination



- Group leader
- Lab sessions
- Regular meeting

Projects: Techniques for Partitioning Tasks

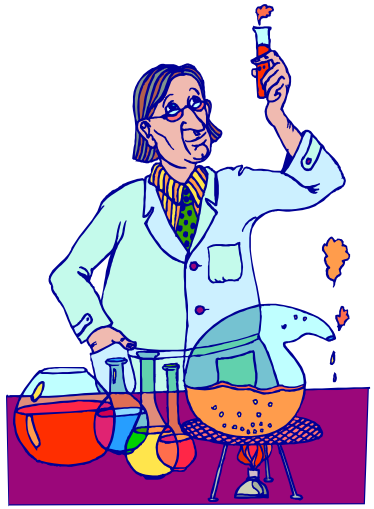
- Functional
  - Person A implements threads, Person B implements semaphores, Person C implements locks…
- Task
  - Person A designs, Person B writes code, Person C tests
  - May be difficult to find right balance, but can focus on each person's strengths (Theory vs systems hacker)
  - Since Debugging is hard, Microsoft has *two* testers for *each* programmer
- Most CSE-150 project teams are functional, but people have had success with task-based divisions

Suggested Documents for You to Maintain

- Organizational Chart
  - Who is responsible for what task?
- Schedule: What is your anticipated timing?
  - This document is critical!
- Meeting notes
  - Document all decisions
  - You can often cut & paste for the design documents
- Design document: the manual plus performance specs
  - This should be the first document generated and the last one finished.
  - Transform your design document to the final project report.

Test Continuously

- Integration tests all the time, not at 11pm on due date!
  - Schedule periodic integration tests
    - » Get everyone in the same room, check out code, build, and test.
    - » Don't wait until it is too late!
- Test early, test later, test again
  - Tendency is to test once and forget; what if something changes in some other part of the code?



Today

- Continue with Synchronization Abstractions
  - Semaphores, monitors, and condition variables

Where are we going with synchronization?

Programs	**Shared Programs**
Higher-level API	**Locks Semaphores Monitors Send/Receive**
Hardware	**Load/Store Disable Ints Test&Set Comp&Swap**

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

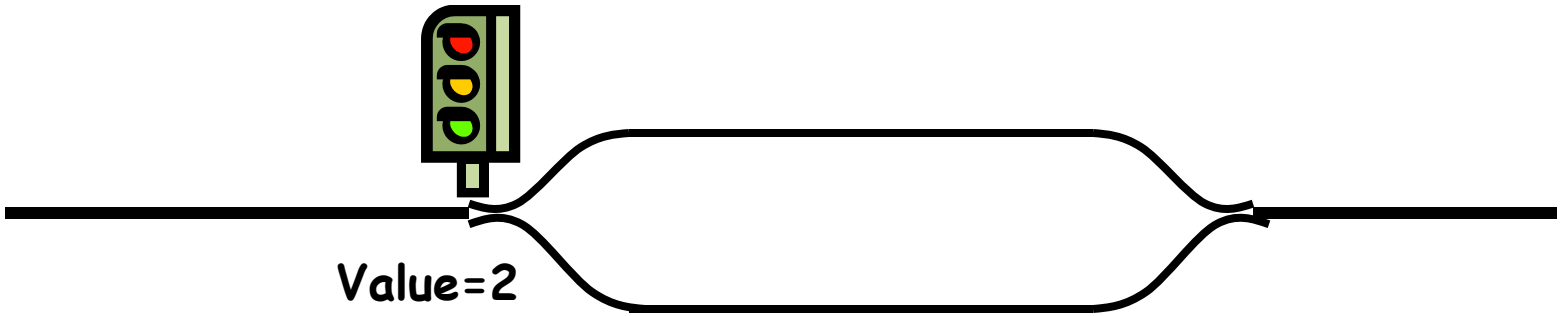
Semaphores

- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX

- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
  - V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
  - Note that P() stands for "*proberen*" (to test) and V() stands for "*verhogen*" (to increment) in Dutch

Semaphores Like Integers Except

- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are P and V – can't read or write value, except to set it initially
  - Operations must be atomic
    - » Two P's together can't decrement value below zero
    - » Thread going to sleep in P won't miss wakeup from V
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



Two Uses of Semaphores

- Mutual Exclusion (initial value = 1)
  - Also called "Binary Semaphore".
  - Can be used for mutual exclusion:

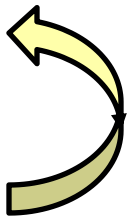
```
semaphore.P();
// Critical section goes here
semaphore.V();
```

- Scheduling Constraints (initial value = 0)
  - Allow thread 1 to wait for a signal from thread 2, i.e., thread 2 schedules thread 1 when a given constraint is satisfied
  - Example: suppose you had to implement ThreadJoin which must wait for thread to terminate:

```
Initial value of semaphore = 0

ThreadJoin {                    // Main program
    semaphore.P();
}

ThreadFinish {                  // Child Thread
    semaphore.V();
}
```



Producer-consumer with a bounded buffer

- Problem Definition
  - Producer puts things into a shared buffer
  - Consumer takes them out
  - Need synchronization to coordinate producer/consumer

- Put a fixed-size buffer between producer and consumer
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty

- Example: Coke machine
  - Producer can put limited number of cokes in machine
  - Consumer can't take cokes out if machine is empty
  - They cannot work at the same time.

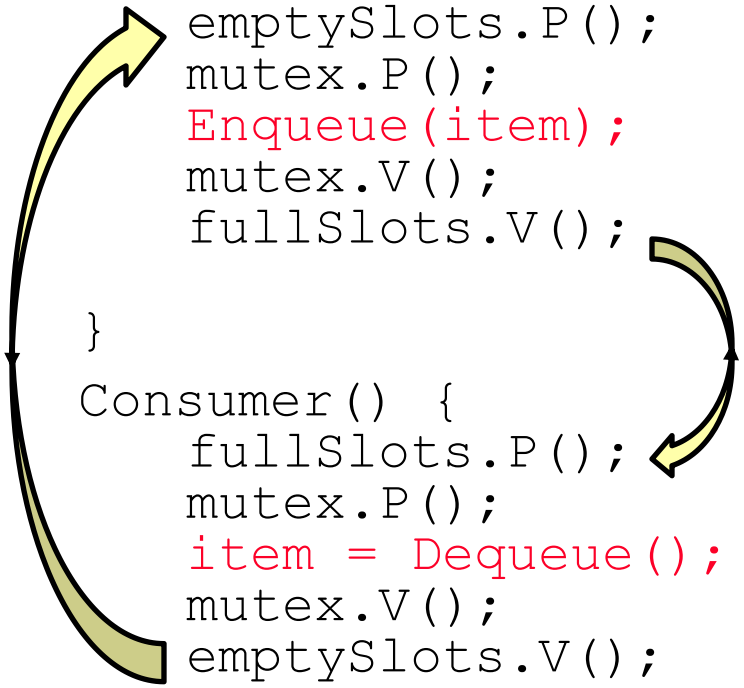
Correctness constraints for solution

- Correctness Constraints:
  - Consumer must wait for producer to fill buffer, if none (scheduling constraint)
  - Producer must wait for consumer to make room in buffer, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are stupid
  - Imagine if in real life: the delivery person is filling the machine and somebody comes up and tries to stick their money into the machine
- General rule of thumb:
  Use a separate semaphore for each constraint
  - `Semaphore fullSlots; // consumer's constraint`
  - `Semaphore emptySlots;// producer's constraint`
  - `Semaphore mutex;     // mutual exclusion`

Full Solution to Bounded Buffer

```
Semaphore fullSlots = 0;   // Initially, no coke
Semaphore emptySlots = bufSize;
                           // Initially, num empty slots
Semaphore mutex = 1;       // No one using machine

Producer(item) {
    emptySlots.P();        // Wait until space
    mutex.P();             // Wait until machine free
    Enqueue(item);
    mutex.V();
    fullSlots.V();         // Tell consumers there is
                           // more coke
}
Consumer() {
    fullSlots.P();         // Check if there's a coke
    mutex.P();             // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptySlots.V();        // tell producer need more
    return item;
}
```



Discussion about Solution

- Is order of P's important?
  - Yes! May cause deadlock when the buffer is full and a producer comes first.

```
Producer(item) {
    mutex.P();
    emptySlots.P();
    Enqueue(item);
    mutex.V();
    fullSlots.V();
}

Consumer() {
    fullSlots.P();
    mutex.P();
    item = Dequeue();
    mutex.V();
    emptySlots.V();
    return item;
}
```

Discussion about Solution

- Is order of P's important?
  - Yes! May cause deadlock when the buffer is empty.

```
Producer(item) {
    emptySlots.P();
    mutex.P();
    Enqueue(item);
    mutex.V();
    fullSlots.V();
}

Consumer() {
    mutex.P();
    fullSlots.P();
    item = Dequeue();
    mutex.V();
    emptySlots.V();
    return item;
}
```

Discussion about Solution

- Is order of P's important?
  - Yes! May cause deadlock when the buffer is full or empty.

```
Producer(item) {
    mutex.P();
    emptySlots.P();
    Enqueue(item);
    mutex.V();
    fullSlots.V();
}
Consumer() {
    mutex.P();
    fullSlots.P();
    item = Dequeue();
    mutex.V();
    emptySlots.V();
    return item;
}
```

Discussion about Solution

- Is order of V's important?
  - No, except that it might affect scheduling efficiency, when the buffer is empty and the consumer comes first.

```
Producer(item) {
    emptySlots.P();
    mutex.P();
    Enqueue(item);
    fullSlots.V();
    mutex.V();
}

Consumer() {
    fullSlots.P();
    mutex.P();
    item = Dequeue();
    mutex.V();
    emptySlots.V();
    return item;
}
```

Discussion about Solution

- Is order of V's important?
  - No, except that it might affect scheduling efficiency, when the buffer is full.

```
Producer(item) {
    emptySlots.P();
    mutex.P();
    Enqueue(item);
    mutex.V();
    fullSlots.V();
}

Consumer() {
    fullSlots.P();
    mutex.P();
    item = Dequeue();
    emptySlots.V();
    mutex.V();
    return item;
}
```

Motivation for Monitors and Condition Variables

- Semaphores are a huge step up; just think of trying to do the bounded buffer with only loads and stores

- Problem is that semaphores are low-level:
  - They are used for both mutex and scheduling constraints
  - How do you prove correctness to someone?
    - » Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious.

- Need a cleaner idea …

Motivation for Monitors and Condition Variables

- Cleaner idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints

- Monitor: a lock and zero or more condition variables for managing concurrent access to shared data
  - Some languages like Java provide this natively – an object (instance of a class) whose methods are implemented with mutual exclusion (using "synchronized" keyword)
    - » To learn how to achieve synchronization using monitor, we cannot use this keyword in our projects. We need to define our own lock and condition variables.
  - Most others use actual locks and condition variables

Simple Monitor Example (version 1)

- Here is an (infinite) synchronized queue

```
Lock lock;
Queue queue;                        // shared data


AddToQueue(item) {
   lock.Acquire();                  // Get Lock
   queue.enqueue(item);             // Add item
   lock.Release();                  // Release Lock
}


RemoveFromQueue() {
   lock.Acquire();                  // Get Lock
   item = queue.dequeue();          // Get next item
   lock.Release();                  // Release Lock
   return(item);                    // Might return null
}
```

- Not very interesting use of "Monitor"
  - Only uses a lock and no condition variables
  - Cannot put consumer to sleep if nothing to dequeue!

Condition Variables

- How do we change the RemoveFromQueue() routine to wait until something is on the queue?
- Condition variable: a variable `x` that implements:
  - `x.wait()`: Wait on a condition (go to sleep?)
  - `x.signal()`: Wake up one waiter, if any
  - Many threads can call `x.wait()`, they will be queued up waiting for a call to `x.signal()`. That call will start the first waiting thread.
- Some systems also implement:
  - `broadcast()` to wake up all waiting threads
- To support sleeping while waiting inside critical section, we add
  - `x.wait(&lock)`: Atomically release the lock and go to sleep. Re-acquire lock later, before returning.
- Rule: Must hold lock when doing condition variable operations

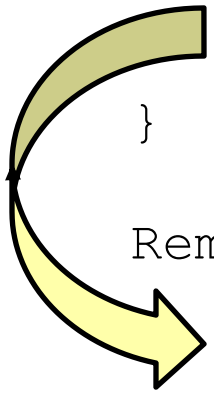
Complete Monitor Example (condition variable)

- Here is an (infinite) synchronized queue

```
Lock lock;
Condition dataready;
Queue queue;                        // shared data


AddToQueue(item) {
   lock.Acquire();                  // Get Lock
   queue.enqueue(item);             // Add item
   dataready.signal();              // Signal any waiters
   lock.Release();                  // Release Lock
}


RemoveFromQueue() {
   lock.Acquire();                  // Get Lock
   while (queue.isEmpty()) {
      dataready.wait(&lock); // If nothing, sleep
   }
   item = queue.dequeue();          // Get next item
   lock.Release();                  // Release Lock
   return(item);
}
```



Summary

- Semaphores: Like integers with restricted interface
  - Two operations:
    - » `P()`: Wait if zero; decrement when becomes non-zero
    - » `V()`: Increment and wake a sleeping task (if exists)
    - » Can initialize value to any non-negative value
  - Use separate semaphore for each constraint
- Monitors: A lock plus one or more condition variables
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
    - » Three Operations: `Wait()`, `Signal()`, and `Broadcast()`

Next

- Reading: Chapter 7.