

CS162
Operating Systems and
Systems Programming
Lecture 5
(extra)

Semaphores, Conditional Variables

February 5, 2014

Anthony D. Joseph

<http://inst.eecs.berkeley.edu/~cs162>

Better Implementation of Locks by Disabling Interrupts

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

```
int value = FREE;
```



```
Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait queue;  
        Go to sleep();  
        // Enable interrupts?  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```

```
Release() {  
    disable interrupts;  
    if (anyone on wait queue) {  
        take thread off wait queue  
        Put at front of ready queue  
    } else {  
        value = FREE;  
    }  
    enable interrupts;  
}
```

Examples of Read-Modify-Write

- `test&set (&address) { /* most architectures */
 result = M[address];
 M[address] = 1;
 return result;
}`
- `swap (&address, register) { /* x86 */
 temp = M[address];
 M[address] = register;
 register = temp;
}`
- `compare&swap (&address, reg1, reg2) { /* 68000 */
 if (reg1 == M[address]) {
 M[address] = reg2;
 return success;
 } else {
 return failure;
 }
}`

Better Locks using test&set

- Key idea: only busy-wait to atomically check lock value

```
int guard = 0;  
int value = FREE;
```



```
Acquire() {  
    // Short busy-wait time  
    while (test&set(guard));  
    if (value == BUSY) {  
        put thread on wait queue;  
        go to sleep() & guard = 0;  
    } else {  
        value = BUSY;  
        guard = 0;  
    }  
}
```

```
Release() {  
    // Short busy-wait time  
    while (test&set(guard));  
    if anyone on wait queue {  
        take thread off wait queue  
        Place on ready queue;  
    } else {  
        value = FREE;  
    }  
    guard = 0;  
}
```

- Note: sleep has to be sure to reset the guard variable

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

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 **constrained** is satisfied
 - Example: suppose you had to implement ThreadJoin which must wait for thread to terminate:

```
Initial value of semaphore = 0  
ThreadJoin {  
    semaphore.P();  
}  
ThreadFinish {  
    semaphore.V();  
}
```

Full Solution to Bounded Buffer

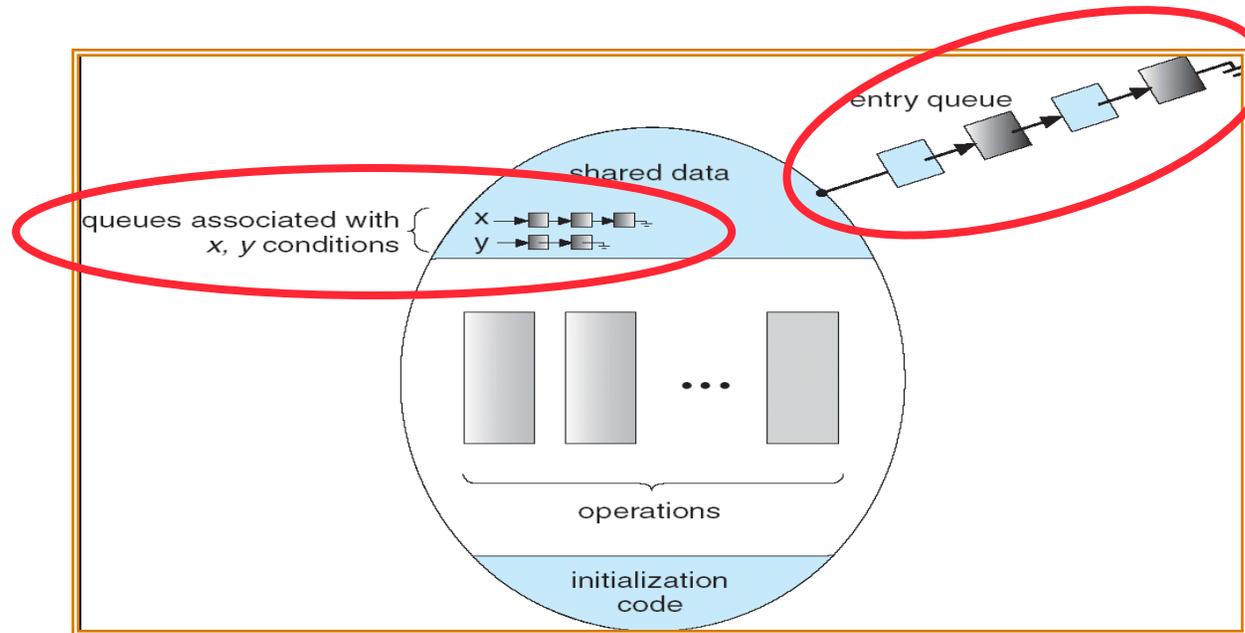
```
Semaphore fullSlots = 0; // Initially, no coke
Semaphore emptySlots = bufSize;
                               // Initially, num empty slots
Semaphore mutex = 1; // No one using machine

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;
}
```

5min Break

Monitor with Condition Variables



- **Lock**: the lock provides mutual exclusion to shared data
 - Always acquire before accessing shared data structure
 - Always release after finishing with shared data
 - Lock initially free
- **Condition Variable**: a queue of threads waiting for something *inside* a critical section
 - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep

Condition Variables

- **Condition Variable**: a queue of threads waiting for something *inside* a critical section
 - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
 - Contrast to semaphores: Can't wait inside critical section
- Operations:
 - `Wait(&lock)`: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
 - `Signal()`: Wake up one waiter, if any
 - `Broadcast()`: Wake up all waiters
- Rule: Must hold lock when doing condition variable ops!

Complete Monitor Example

- Here is an (infinite) synchronized queue

```
Lock lock;
Condition dataready;
Queue queue;

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);
}
```